Events

This lesson talks about Events, which is a lesser known utility class within the threading module.

Events

An event object is one of the simplest primitives available for synchronization. Internally, it has a boolean flag that can be set or unset using the methods **`set()`** and **`clear()`**. Additionally, a thread can check if the flag is set to true by invoking the **`is_set()`** method.

The event object exposes a **`wait()`** method that threads can invoke to wait for the internal boolean flag to become true. If the flag is already true, the thread returns immediately. If there are multiple threads waiting on the event object and an active thread sets the flag then all the waiting threads are unblocked.

`Event` is a convenience class and a wrapper over a condition variable with a boolean predicate. This is the most common setup for many cooperating threads where two or more threads coordinate among themselves on a boolean predicate.

Differences with Semaphores

Event objects may seem similar to semaphore or a bounded semaphore but there are slight differences:

- An unbounded semaphore can have its internal counter incremented as many times as **`acquire()`** is invoked on it, whereas an event object maintains an internal boolean flag that can only flip between two state: set or unset.

- Can a bounded semaphore intialized to 1 be equivalent to an event object? The answer is no because the bounded semaphore will raise a **`ValueError`** if the bounded semaphore is acquired more number of times than the initial passed in capacity. Also acquiring a semaphore decrements the internal counter of the semaphore whereas waiting on an event object doesn't change the state of the internal boolean flag.

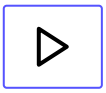
- A thread never gets blocked on **`wait()`** of an event object if the internal flag is set to true no matter how many times the thread invokes the **`wait()`** method.

Below is a naive example of using event to coordinate interaction between two threads:

Event example

```
from threading import Thread
from threading import Event
import time


def task1():
    event.wait()
    event.wait()
    event.wait()
    print("thread invoked wait thrice")


def task2():
    time.sleep(1)
    event.set()


event = Event()

thread1 = Thread(target=task1)
thread1.start()

thread2 = Thread(target=task2)
thread2.start()

thread1.join()
thread2.join()
```


```
from threading import Thread
from threading import Event
import time


def task1():
    event.wait()
    event.wait()
    event.wait()
    print("thread invoked wait() thrice")


def task2():
    time.sleep(1)
    event.set()


event = Event()

thread1 = Thread(target=task1)
thread1.start()

thread2 = Thread(target=task2)
thread2.start()

thread1.join()
thread2.join()
```




We can implement our prime printing program using events. The code appears below:

Prime printing program using events

```
from threading import Thread
from threading import Event
import time


def printer_thread():
    global primeHolder

    while not exitProg:
        # wait for a prime number to become available
        prime_available.wait()

        # print the prime number
        print(primeHolder)
        primeHolder = None

        # reset the event to false
        prime_available.clear()

        # let the finder thread know that printing is done
        prime_printed.set()


def is_prime(num):
    if num == 2 or num == 3:
        return True

    div = 2

    while div <= num / 2:
        if num % div == 0:
            return False
        div += 1
    return True


def finder_thread():
    global primeHolder
```

```
    i = 1

    while not exitProg:

        while not is_prime(i):
            i += 1
            # Add a timer to slow down the thread
            # so that we can see the output
            time.sleep(.01)

        primeHolder = i

        # let the printer thread know we have
        # a prime available for printing
        prime_available.set()

        # wait for printer thread to print the prime
        prime_printed.wait()

        # reset the flag
        prime_printed.clear()

        i += 1



prime_available = Event()
prime_printed = Event()
primeHolder = None
exitProg = False

printerThread = Thread(target=printer_thread)
printerThread.start()

finderThread = Thread(target=finder_thread)
finderThread.start()

# Let the threads run for 3 seconds
time.sleep(3)
```

```
exitProg = True

prime_available.set()
prime_printed.set()

printerThread.join()
finderThread.join()
```

```
from threading import Thread
from threading import Event
import time


def printer_thread():
    global primeHolder

    while not exitProg:
        # wait for a prime number to become available
        prime_available.wait()

        # print the prime number
        print(primeHolder)
        primeHolder = None

        # reset the event to false
        prime_available.clear()

        # let the finder thread know that printing is done
        prime_printed.set()


def is_prime(num):
    if num == 2 or num == 3:
        return True

    div = 2

    while div <= num / 2:
        if num % div == 0:
            return False
        div += 1
    return True


def finder_thread():
    global primeHolder

    i = 1

    while not exitProg:

        while not is_prime(i):
            i += 1
            # Add a timer to slow down the thread
            # so that we can see the output
            time.sleep(.01)

        primeHolder = i
```

```
        # let the printer thread know we have
        # a prime available for printing

        prime_available.set()

        # wait for printer thread to print the prime
        prime_printed.wait()

        # reset the flag
        prime_printed.clear()

        i += 1



prime_available = Event()
prime_printed = Event()
primeHolder = None
exitProg = False

printerThread = Thread(target=printer_thread)
printerThread.start()

finderThread = Thread(target=finder_thread)
finderThread.start()

# Let the threads run for 3 seconds
time.sleep(3)

exitProg = True
prime_available.set()
prime_printed.set()

printerThread.join()
finderThread.join()
```

Back

Semaphores

Next

Timer

Mark as Completed

Report an Issue

Ask a Question (https://discuss.educative.io/tag/events__threading-module__python-concurrency-for-senior-engineering-interviews)