Semaphores

This lesson introduces semaphores and their uses in Python.

Semaphores

Semaphore is one of the oldest synchronization primitives, invented by Edsger Dijkstra (https://en.wikipedia.org/wiki/Edsger_W._Dijkstra). A semaphore is nothing more than an atomic counter that gets decremented by one whenever **`acquire()`** is invoked and incremented by one whenever **`release()`** is called. The semaphore can be initialized with an initial count value. If none is specified, the semaphore is initialized with a value of one.

Creating semaphore

```
# semaphore initialized with a default count of 1
semaphore = Semaphore()

# semaphore initialized with count set to 5
sem_with_count = Semaphore(5)
```

acquire ( )

If a thread invokes **`acquire()`** on a semaphore, the semaphore counter

is decremented by one. If the count is greater than 0, then the thread immediately returns from the **`acquire()`** call. If the semaphore counter is zero when a thread invokes **`acquire()`**, the thread gets blocked till another thread releases the semaphore.

release ( )

When a thread invokes the **`release()`** method, the internal semaphore counter is incremented by one. If the counter value is zero and another thread is already blocked on an **`acquire()`** then a release would unblock the waiting thread. If multiple threads are blocked on the semaphore, then one thread is arbitrarily chosen.

Using Semaphores

Semaphores can be used in versatile ways. The primary use of semaphores is signaling among threads which are working to achieve a common goal. Consider the below snippet which uses a condition variable between threads.

Missed Signal

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


def task1():
    cond_var.acquire()
    cond_var.wait()
    cond_var.release()


def task2():
    cond_var.acquire()
    cond_var.notify()
    cond_var.release()


cond_var = Condition()

# start thread 2 first which invokes notify
thread2 = Thread(target=task2)
thread2.start()

# delay starting thread 1 by three seconds
time.sleep(3)

# start thread 1
thread1 = Thread(target=task1)
thread1.start()

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

The above program will never exit since the notifying thread notifies before the first thread has a chance to wait on the condition variable. **This is a manifestation of a missed signal.**

You may realize from your reading in the previous sections that the way we are using the condition variable's **`wait()`** method is incorrect. The idiomatic usage of **`wait()`** is in a while loop with an associated boolean condition. For now, observe the possibility of losing signals between threads.

The faulty program appears in the code widget below:

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


def task1():
    cond_var.acquire()
    cond_var.wait()
    cond_var.release()


def task2():
    cond_var.acquire()
    cond_var.notify()
    cond_var.release()


cond_var = Condition()

# start thread 2 first which invokes notify
thread2 = Thread(target=task2)
thread2.start()

# delay starting thread 1 by three seconds
time.sleep(3)

# start thread 1
thread1 = Thread(target=task1)
thread1.start()

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

One way to remedy the above situation is to use semaphores. The internal counter of the semaphore *remembers* how many times the signal was received. Replacing the above code with semaphore looks as follows:

Fixing missed signal

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


def task1():
    sem.acquire()


def task2():
    sem.release()

# initialize with zero
sem = Semaphore(0)

# start thread 2 first which invokes release()
thread2 = Thread(target=task2)
thread2.start()

# delay starting thread 1 by three seconds
time.sleep(3)

# start thread 1
thread1 = Thread(target=task1)
thread1.start()

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

Note that we are initializing the semaphore with a count of zero. If we used the default initialization of 1, then the first call to **`acquire()`** would not block the thread.

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


def task1():
    sem.acquire()


def task2():
    sem.release()

# initialize with zero
sem = Semaphore(0)

# start thread 2 first which invokes release()
thread2 = Thread(target=task2)
thread2.start()

# delay starting thread 1 by three seconds
time.sleep(3)

# start thread 1
thread1 = Thread(target=task1)
thread1.start()

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

Rewriting the prime printer program

As an additional example, we rewrite our prime printer program from previous sections using semaphores. Note how the code becomes less

verbose and possibly easier to follow:

Printer Program

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


def printer_thread():
    global primeHolder

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

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

        # let the finder thread find the next prime
        sem_print.release()


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
        sem_find.release()

        # wait for printer thread to complete
        # printing the prime number
        sem_print.acquire()

        i += 1


sem_find = Semaphore(0)
sem_print = Semaphore(0)
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

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

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


def printer_thread():
    global primeHolder

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

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

        # let the finder thread find the next prime
        sem_print.release()


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
        sem_find.release()
```

```
        # wait for printer thread to complete

        # printing the prime number
        sem_print.acquire()

        i += 1


sem_find = Semaphore(0)
sem_print = Semaphore(0)
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

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

Back

Next

... continued

Events

Mark as Completed

Report an Issue

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