Condition Variables

This lesson explains the concept of condition variables in Python.

Condition Variables

Synchronization mechanisms need more than just mutual exclusion; a general need is to be able to wait for another thread to do something. Condition variables provide mutual exclusion and the ability for threads to wait for a predicate to become true.

For Java programmers, the condition variable and its working may seem eerily familiar and rightly so because the threading module borrows a lot from Java's concurrency architecture. We'll examine the similarities towards the end of the lesson. First, let's try to understand why we need condition variables in the first place.

Why

In the previous sections we worked with locks which are used to enforce serial access to shared state. However, locks aren't enough when threads want to coordinate among themselves. Imagine a scenario where we have two threads working together to find prime numbers and print them. Say the first thread finds the prime number and the second thread is responsible for printing the found prime. The first thread (finder) sets a boolean flag whenever it determines an integer is a prime number. The second (printer) thread *needs to know when the finder thread has hit* *upon a prime number.* The naive approach is to have the printer thread do a busy wait and keep polling for the boolean value. Let's see what this approach looks like:

Approach 1

```
def printer_thread_func():
    global prime_holder
    global found_prime

    while not exit_prog:
        while not found_prime and not exit_prog:
            time.sleep(0.1)

        if not exit_prog:
            print(prime_holder)

            prime_holder = None
            found_prime = False


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_func():
    global prime_holder
    global found_prime

    i = 1

    while not exit_prog:

        while not is_prime(i):
            i += 1

        prime_holder = i
        found_prime = True

        while found_prime and not exit_prog:
            time.sleep(0.1)

        i += 1


found_prime = False
prime_holder = None
exit_prog = False

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

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

# Let the threads run for 5 seconds
time.sleep(3)

# Let the threads exit
exit_prog = True

printer_thread.join()
finder_thread.join()
```

```
from threading import Thread
import time


def printer_thread_func():
    global prime_holder
    global found_prime

    while not exit_prog:
        while not found_prime and not exit_prog:
            time.sleep(0.1)

        if not exit_prog:
            print(prime_holder)

            prime_holder = None
            found_prime = False


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_func():
    global prime_holder
    global found_prime

    i = 1

    while not exit_prog:

        while not is_prime(i):
            i += 1

        prime_holder = i
        found_prime = True

        while found_prime and not exit_prog:
            time.sleep(0.1)

        i += 1
```

```
found_prime = False
prime_holder = None
exit_prog = False

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

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

# Let the threads run for 5 seconds
time.sleep(3)

# Let the threads exit
exit_prog = True

printer_thread.join()
finder_thread.join()
```

Notice we aren't using any synchronization primitives in the program above. Not even locks to guard writes to shared variables. The program goes through two states, finding primes and then printing them, and repeats the sequence. The above program would work because there are only two threads involved but would fail with greater than two threads. Folks with Java background would realize that a similar program in Java wouldn't work given the memory model of Java, and would likely require the shared variables to be marked volatile.

The above program is essentially a producer-consumer problem. The printer thread is a consumer and the finder thread is a producer. The printer thread needs to be *signaled* somehow that a prime number has been discovered for it to print. Do you see a condition here? The condition in our program is the discovery of the prime number represented by the boolean variable **`found_prime`**. Realize that locks don't help us signal other threads when a condition becomes true.

One shortcoming of the above code is we have the printer thread constantly polling in a while loop for the **`found_prime`** variable to become true. This is called busy waiting and is highly discouraged as it unnecessarily wastes CPU cycles. Ideally, the printer thread should go to sleep so that it doesn't consume any system resources and be woken up when the condition it needs to act upon becomes true. This can be achieved through condition variables.

In the next section, we'll see how to use condition variables.

← Back
RLock

Next →
... continued

✓ Mark as Completed

Report an Issue

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