... continued

Continuation of discussion on condition variables.

Let's rewrite our printer thread first using the correct condition variable pattern. The first task the printer does is to wait for a prime number to become available. We can achieve that using the following snippet:

Printer thread waiting

```
        cond_var.acquire()
        while not found_prime and not exit_prog:
            cond_var.wait()
        cond_var.release()
```

Once the printer thread has printed the found prime number, it needs to let the finder thread know to continue finding the next prime number. To achieve that we'll notify the condition variable after printing the prime number. In the finder thread code, we'll make the finder thread wait on the condition variable. The complete code for the printer thread appears below:

Printer thread

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

    while not exit_prog:

        # wait for a prime number to become
        # available for printing
        cond_var.acquire()
        while not found_prime and not exit_prog:
            cond_var.wait()
        cond_var.release()

        if not exit_prog:
            # print the prime number
            print(prime_holder)

            # reset. We can skip this statement if we like
            prime_holder = None

            # make sure to wake up the finder thread
            cond_var.acquire()
            found_prime = False
            cond_var.notify()
            cond_var.release()
```

Finder Thread

Now we'll turn our attention to the finder thread. It first attempts to find a prime number. Once found, it needs to communicate to the printer thread that a prime number is ready for printing. It does that by signaling the condition variable as follows:

Finder thread signaling printer thread

```
        cond_var.acquire()
        found_prime = True
        cond_var.notify()
        cond_var.release()
```

The next step for the finder thread is to wait for the printer thread to complete its printing. The finder thread simply waits on the same condition variable for the printer thread to signal. The complete code for the finder thread appears below:

Finder thread

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

    i = 1

    while not exit_prog:

        while not is_prime(i):
            i += 1

        primeHolder = i

        cond_var.acquire()
        found_prime = True
        cond_var.notify()
        cond_var.release()

        cond_var.acquire()
        while found_prime and not exit_prog:
            cond_var.wait()
        cond_var.release()

        i += 1
```

Main Thread

Let's not forget the main thread which spawns both the printer and the finder threads. Note that the variable **`exit_prog`** is solely to control how long our program runs but doesn't affect the core working of the two spawned threads. The main thread needs to signal the condition variable too because it is possible that one of the threads ends up

waiting on the condition variable while the other thread exits its while loop, causing the program to hang. The code for the main thread appears below:

Finder thread

```
cond_var = Condition()
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 3 seconds
time.sleep(3)

# Let the threads exit
exit_prog = True

cond_var.acquire()
cond_var.notifyAll()
cond_var.release()

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

The complete code for the program appears below:

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


def printer_thread_func():
    global prime_holder
    global found_prime

    while not exit_prog:

        cond_var.acquire()
        while not found_prime and not exit_prog:
            cond_var.wait()
        cond_var.release()

        if not exit_prog:
            print(prime_holder)

            prime_holder = None

            cond_var.acquire()
            found_prime = False
            cond_var.notify()
            cond_var.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_func():
    global prime_holder
    global found_prime

    i = 1

    while not exit_prog:

        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)

        prime_holder = i

        cond_var.acquire()
        found_prime = True
        cond_var.notify()
        cond_var.release()

        cond_var.acquire()
        while found_prime and not exit_prog:
            cond_var.wait()
        cond_var.release()

        i += 1


cond_var = Condition()
found_prime = False
prime_holder = None
exit_prog = False

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

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

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

# Let the threads exit
exit_prog = True

cond_var.acquire()
cond_var.notifyAll()
cond_var.release()

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

Question

Can you explain what the consequences would be if we moved the `cond_var` associated lock's acquire and release inside the while loop as follows:

```
        while not found_prime and not exit_prog:
            cond_var.acquire()
            cond_var.wait()
            cond_var.release()
```

Q

A) Program would function correctly at all times

B) Program would deadlock in some scenarios

C) Program would always deadlock.

Submit Answer

Reset Quiz

Question

In the prime printer program, we have used the following snippet to flip the value of the variable `found_prime`:

```
0.    cond_var.acquire()
1.    found_prime = <new_value> # can be True or False
2.    cond_var.notify()
3.    cond_var.release()
```

Do you think the program would work correctly if we switched the order of lines 1 and 2 as follows, i.e. we `notify()` first and then flip the value of the variable?

```
0.    cond_var.acquire()
1.    cond_var.notify()
2.    found_prime = <new_value> # can be True or False
3.    cond_var.release()
```

Q

A) Yes

B) No

Submit Answer

Reset Quiz

Question

Can we use a different condition variable to signal the finder thread?

Indeed, another way to implement the prime printer program is to use an additional condition variable. However, the logic of the program stays the same. The implementation is provided below:

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

    while not exit_prog:

        cond_var.acquire()
        while not found_prime and not exit_prog:
            cond_var.wait()
        cond_var.release()

        if not exit_prog:
            print(prime_holder)

            prime_holder = None

            another_cond_var.acquire()
            found_prime = False
            another_cond_var.notify()
            another_cond_var.release()
    print("printer exiting")


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 prime_holder
    global found_prime


    i = 1

    while not exit_prog:

        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)

        prime_holder = i

        cond_var.acquire()
        found_prime = True
        cond_var.notify()
        cond_var.release()

        another_cond_var.acquire()
        while found_prime and not exit_prog:
            another_cond_var.wait()
        another_cond_var.release()

        i += 1
    print("finder exiting")
```

The resulting program is however more brittle and prone to bugs in case of future modifications. The reason is that we are now modifying the `found_prime` variable within two condition variables, and that doesn't guarantee that the changes to the `found_prime` variable are being made by a single thread at a time. That guarantee now comes from the structure/logic of the code that `found_prime` is indeed modified by a single thread at any point. Any change in the logic/structure can violate the guarantee and cause hard to find bugs. This is an example of bad design and implementation.

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


def printer_thread():
    global prime_holder
    global found_prime

    while not exit_prog:

        cond_var.acquire()
        while not found_prime and not exit_prog:
            cond_var.wait()
        cond_var.release()

        if not exit_prog:
            print(prime_holder)

            prime_holder = None

            another_cond_var.acquire()
            found_prime = False
            another_cond_var.notify()
            another_cond_var.release()
    print("printer exiting")


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 prime_holder
    global found_prime

    i = 1

    while not exit_prog:

        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)

        prime_holder = i

        cond_var.acquire()
        found_prime = True
        cond_var.notify()
        cond_var.release()

        another_cond_var.acquire()
        while found_prime and not exit_prog:
            another_cond_var.wait()
        another_cond_var.release()

        i += 1
    print("finder exiting")


if __name__ == "__main__":
    cond_var = Condition()
    another_cond_var = Condition()
    found_prime = False
    prime_holder = None
    exit_prog = False


    exit_prog = 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)

    exit_prog = True

    # Let the threads exit
    cond_var.acquire()
    cond_var.notifyAll()
    cond_var.release()

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

Question

Imagine business requirements dictate that we use two printer threads instead of one in the prime printer program and each printer thread prints a unique prime number, i.e. the same prime number isn't printed twice. Can you think of the changes required to make the change?

The major change would happen in the printer function, where we need to guarantee that a single printer thread prints a prime number and then flips the `found_prime` variable. Also, realize that we'll need to use `notifyAll()` because of multiple threads.

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

    while not exit_prog:

        cond_var.acquire()
        while not found_prime and not exit_prog:
            cond_var.wait()

        if not exit_prog:
            print("{0} prints {1}".format(current_thread().get
Name(), prime_holder))

            prime_holder = None

            found_prime = False
            # use notifyAll() instead of notify()
            cond_var.notifyAll()
        cond_var.release()

    print("printer {0} exiting".format(current_thread().getNam
e()))
```

Notice that we have moved the **`cond_var.release()`** statement at the end of the while loop. The condition variable's lock is held all the way from the **`cond_var.wait()`** statement to the statement **`cond_var.release()`** ensuring that only a single printer thread prints a prime.

The finder thread doesn't need any change. We can use **`notifyAll()`** instead of **`notify()`**, but since we only want one printer thread to be active it doesn't change the logic of the program. In case of **`notifyAll()`**, only one printer thread will move ahead while the rest will go back to waiting because of the while loop. The complete code appears in the code widget below:

code widget below:

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


def printer_thread():
    global prime_holder
    global found_prime

    while not exit_prog:

        cond_var.acquire()
        while not found_prime and not exit_prog:
            cond_var.wait()

        if not exit_prog:
            print("{0} prints {1}".format(current_thread().getName(), prime_holder)

            prime_holder = None

            found_prime = False
            cond_var.notifyAll()
        cond_var.release()

    print("printer {0} exiting".format(current_thread().getName()))


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 prime_holder
    global found_prime

    i = 1

    while not exit_prog:

        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)

        prime_holder = i

        cond_var.acquire()
        found_prime = True
        cond_var.notify()
        cond_var.release()

        cond_var.acquire()
        while found_prime and not exit_prog:
            cond_var.wait()
        cond_var.release()

        i += 1

    print("finder exiting")


if __name__ == "__main__":
    cond_var = Condition()
    found_prime = False
    prime_holder = None
    exit_prog = False


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

    printerThread2 = Thread(target=printer_thread)
    printerThread2.start()

    printerThread3 = Thread(target=printer_thread)
    printerThread3.start()

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

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

    exit_prog = True

    # Let the threads exit
    cond_var.acquire()
    cond_var.notifyAll()
    cond_var.release()

    printerThread.join()
    printerThread2.join()
```

```
printerThread2.join()
printerThread3.join()
finderThread.join()
```

Examine the output of the above program and notice that only a single thread ever prints all the primes while the other threads keep sleeping. This is possibly because of how Python implements the **`notify()`** call. If you change **line#62** in the code widget to **`notifyAll()`** you'll see the output show all three printer threads participating in printing the prime numbers.

Last but not the least, understand that in case of the printer program we can't use the **`notify()`** call even though one may incorrectly reason that we are attempting to wake up only the finder thread. The issue is that both the printer and finder threads are waiting on the same condition variable and a **`notify()`** call may wake up the printer thread and will go back to waiting since the finder thread hasn't found a new prime causing a deadlock. However, if the **`notify()`** call wakes up the finder thread instead, the program would function correctly for one more iteration.

Back

... continued

Next

... continued

Mark as Completed

Report an Issue

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