... continued

Continuation of discussion on condition variables.

We create a condition variable as follows:

Creating a condition variable

```
cond_var = Condition()
```

The two important methods of a condition variable are:

- **`wait()`** - invoked to make a thread sleep and give up resources
- **`notify()`** - invoked by a thread when a condition becomes true and the invoking threads want to inform the waiting thread or threads to proceed

A condition variable is always associated with a lock. The lock can be either reentrant or a plain vanilla lock. **The associated lock must be acquired before a thread can invoke `wait()` or `notify()` on the condition variable.**

Incorrect way of using condition variables

```
cond_var = Condition()
cond_var.wait()  # throws an error
```

```
from threading import Condition
```

```
cond_var = Condition()
cond_var.wait()  # throws an error
```

Incorrect Usage of Condition Variable

We can create a lock ourselves and pass it to the condition variable's constructor. If no lock object is passed then a lock is created underneath the hood by the condition variable. In the examples below we demonstrate all the different ways of invoking condition variables.

We can create the condition variable as follows:

Creating a condition variable by passing a custom lock

```
lock = Lock()
cond_var = Condition(lock) # pass custom lock to condition var
iable
cond_var.acquire()
cond_var.wait()
```

We can also create a condition variable without passing in a custom lock.

Creating a condition variable without passing a lock

```
cond_var = Condition()
cond_var.acquire()
cond_var.wait()
```

Given what we know so far about condition variables, let's try to rewrite the printer thread code from the previous section. Note, that we are checking for the boolean variables in an `if` statement.

Printer thread refactored code

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

def printer_thread_func():
    global prime_holder
    global found_prime

    while not exit_prog:

        # check for predicate
        cond_var.acquire()
        if not found_prime and not exit_prog:
            cond_var.wait()
        cond_var.release()


        if not exit_prog:
            print(prime_holder)
            prime_holder = None

            # acquire lock before modifying shared variable
            cond_var.acquire()
            found_prime = False
            # remember to wake up the other thread
            cond_var.notify()
            cond_var.release()
```

Note that we have rid ourselves of the busy wait loop. Now we check if the `found_prime` is false and if so we first acquire the lock on the condition variable and then invoke the `wait()` method on it. When the finder thread invokes `notify()` on the variable `cond_var`, that is the moment when the printer thread will wake up from its slumber and continue.

There are however two questions we need to answer:

- If the printer thread acquires the lock on the condition variable **cond_var** then how can the finder thread **acquire()** the lock when it needs to invoke the **notify()** method?
- Can the condition, which is the variable **found_prime**, change once the printer thread is woken up?

The answer to the first question is that when a thread invokes **wait()** it simultaneously gives up the lock associated with the condition variable. Only when the sleeping thread wakes up again on a **nofity()**, will it reacquire the lock.

The second question is very important and leads us to the correct idiomatic usage of the condition variable. The way we have nested the acquire and wait call under an if statement is incorrect. The reason is that if a thread invokes **notifyAll()** on a condition variable, then all the threads waiting on the condition variable will be woken up but only one thread will be allowed to make progress. Once the first thread exits the critical section and releases the lock associated with the condition variable, another thread, from the set of threads that were waiting when the original **notifyAll()** call was made, is allowed to make progress. This may not be appropriate for every use case and certainly not for ours if we had multiple printer threads. We would want a printer thread to make progress only when the condition **found_prime** is set to true. This can only be possible with a while loop where we check if the condition **found_prime** is true before allowing a printer thread to move ahead. The change will look as follows:

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

On a side note, remember that a woken up thread reacquires the lock associated with the condition variable the thread was waiting on and doesn't give up the lock until it releases the condition variable. Hence, any statements executed between the **`wait()`** and the **`release()`** call would happen while the thread holds the lock assuring mutual exclusion within that block of code. Consider the below pseudocode:

```
0.     cond_var.acquire()
1.     while predicate is not True:
2.         cond_var.wait()
3.
4.     # code statement 1
5.     # code statement 2
6.     # code statement 3
7.
8.     cond_var.release()
```

In the hypothetical snippet above, the **lines 2 to 8** would be executed by a woken up thread while holding the lock associated with the condition variable.

Spurious Wakeups

We may convince ourselves to not use a while loop if we only have a single printer thread in our program. However, a thread can be woken up even if there has been no notification!

A peculiarity of condition variables is the possibility of spurious wakeups. It means that a thread might wakeup as if it has been signaled even though nobody called **`notify()`** on the condition variable in question. This is specifically allowed by the POSIX standard because it allows more efficient implementations of condition variables under some circumstances. Such wakeups are called **spurious wakeups**.

A thread that has been woken up does not imply that the conditions for it to move forward hold. The thread must test the conditions again for validity before moving forward. In conclusion, we must always check for conditions in a loop and **`wait()`** inside it. The correct idiomatic usage of a condition variable appears below:

Idiomatic use of wait()

```
acquire lock
while(condition_to_test is not satisfied):
    wait

# condition is now true, perform necessary tasks

release lock
```

The idiomatic usage to notify() is as follows:

Idiomatic use of notify

```
acquire lock
set condition_to_test to true/satisfied
notify
release lock
```

In the next section, we'll rewrite our printer and finder threads using condition variables.

← Back
Condition Variables

Next →
... continued

✓ Mark as Completed

Ask a Question

Report an Issue

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