... continued

Continuation of discussion on condition variables.

In the previous sections, we worked with **`wait()`** and **`notify()`** methods. These methods have closely related cousins which take in parameters. We discuss them as follows:

wait(n)

The **`wait(n)`** method takes in a floating point parameter *n*. This is the number of seconds a calling thread would wait to be notified by another thread. The wait method times out after n seconds and the thread is woken up even if no notification is received. Consider the below snippet:

Using wait(n)

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

flag = False

cond_var = Condition()


def child_task():
    cond_var.acquire()

    if (flag == False):
        cond_var.wait(1)

    if (flag == False):
        print("child thread times out waiting for a notificati
on")

    # don't forget to release the lock
    cond_var.release()


thread = Thread(target=child_task)
thread.start()

time.sleep(3)
thread.join()
```

Note that we have digressed from the idiomatic usage of the wait method by not testing for the condition in a while loop. Since we don't want the child thread to be stuck in a loop waiting for a condition to become true we are using an if statement. The child thread times out after one second.

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

flag = False

cond_var = Condition()


def child_task():
    cond_var.acquire()

    if (flag == False):
        cond_var.wait(1)

    if (flag == False):
        print("child thread times out waiting for a notification")

    # don't forget to release the lock
    cond_var.release()


thread = Thread(target=child_task)
thread.start()

time.sleep(3)
thread.join()
```

notify_all ( )

`notify_all()` method can be used when there is more than one thread waiting on a condition variable. It can also be used if there's a single thread waiting. The sequence of events on a **`notify_all()`** when multiple threads are waiting is described below:

- A thread comes along acquires the lock associated with the condition variable, and calls **`wait()`**

- The thread invoking **`wait()`** gives up the lock and goes to sleep or is taken off the CPU timeslice

- The given up lock can be reacquired by a second thread that then too calls **`wait()`**, gives up the lock, and goes to sleep.

- Notice that the lock is available for any other thread to acquire and either invoke a wait or a notify on the associated condition variable.

- Another thread comes along acquires the lock and invokes **`notify_all()`** and subsequently releases the lock.

- Note it is imperative to release the lock, otherwise the waiting threads can't reacquire the lock and return from the **`wait()`** call.

- The waiting threads are all woken up but only one of them gets to acquire the lock. This thread returns from the **`wait()`** method and proceeds forward. The thread selected to acquire the lock is random and not in the order in which threads invoked **`wait()`**.

- Once the thread that is the first to wake up and make progress releases the lock, other threads acquire the lock one by one and proceed ahead.

Study the code snippet below, where we initially have three threads wait on a condition variable before **`notify_all()`** is used to wake them all.

notify_all work flow

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

flag = False

cond_var = Condition()


def child_task():
    global flag
    name = current_thread().getName()

    cond_var.acquire()
    if not flag:
        cond_var.wait()
        print("\n{0} woken up \n".format(name))

    cond_var.release()

    print("\n{0} exiting\n".format(name))


thread1 = Thread(target=child_task, name="thread1")
thread2 = Thread(target=child_task, name="thread2")
thread3 = Thread(target=child_task, name="thread3")

thread1.start()
thread2.start()
thread3.start()

cond_var.acquire()
cond_var.notify_all()
cond_var.release()

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

```
thread3.join()


print("main thread exits")
```

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

flag = False

cond_var = Condition()


def child_task():
    global flag
    name = current_thread().getName()

    cond_var.acquire()
    if not flag:
        cond_var.wait()
        print("\n{0} woken up \n".format(name))

    cond_var.release()

    print("\n{0} exiting\n".format(name))


thread1 = Thread(target=child_task, name="thread1")
thread2 = Thread(target=child_task, name="thread2")
thread3 = Thread(target=child_task, name="thread3")

thread1.start()
thread2.start()
thread3.start()

cond_var.acquire()
cond_var.notify_all()
cond_var.release()

thread1.join()
thread2.join()
thread3.join()

print("main thread exits")
```

If you execute the above code multiple times, you'll recognize that the order in which threads are woken up is random.

notify (n)

Another variant of the notify method is the **`notify(n)`** which wakes up n threads. If, say five threads are waiting on a condition variable and we pass invoke **`notify(3)`**, then only three of the five threads will randomly get notified.

Question

Consider an abridged version of the code we discussed in this lesson. The `child_task` method exits without releasing the lock. What would be the outcome of running the program? The changed program is shown below:

```
flag = False

lock = Lock()
cond_var = Condition(lock)


def child_task():
    global flag
    name = current_thread().getName()

    cond_var.acquire()
    while not flag:
        cond_var.wait()
        print("\n{0} woken up \n".format(name))

    print("\n{0} exiting\n".format(name))


if __name__ == "__main__":
    thread1 = Thread(target=child_task, name="thread1")
    thread1.start()

    # give the child task to wait on the condition variable
    time.sleep(1)

    cond_var.acquire()
    flag = True
    cond_var.notify_all()
    cond_var.release()

    thread1.join()
    print("main thread exits")
```

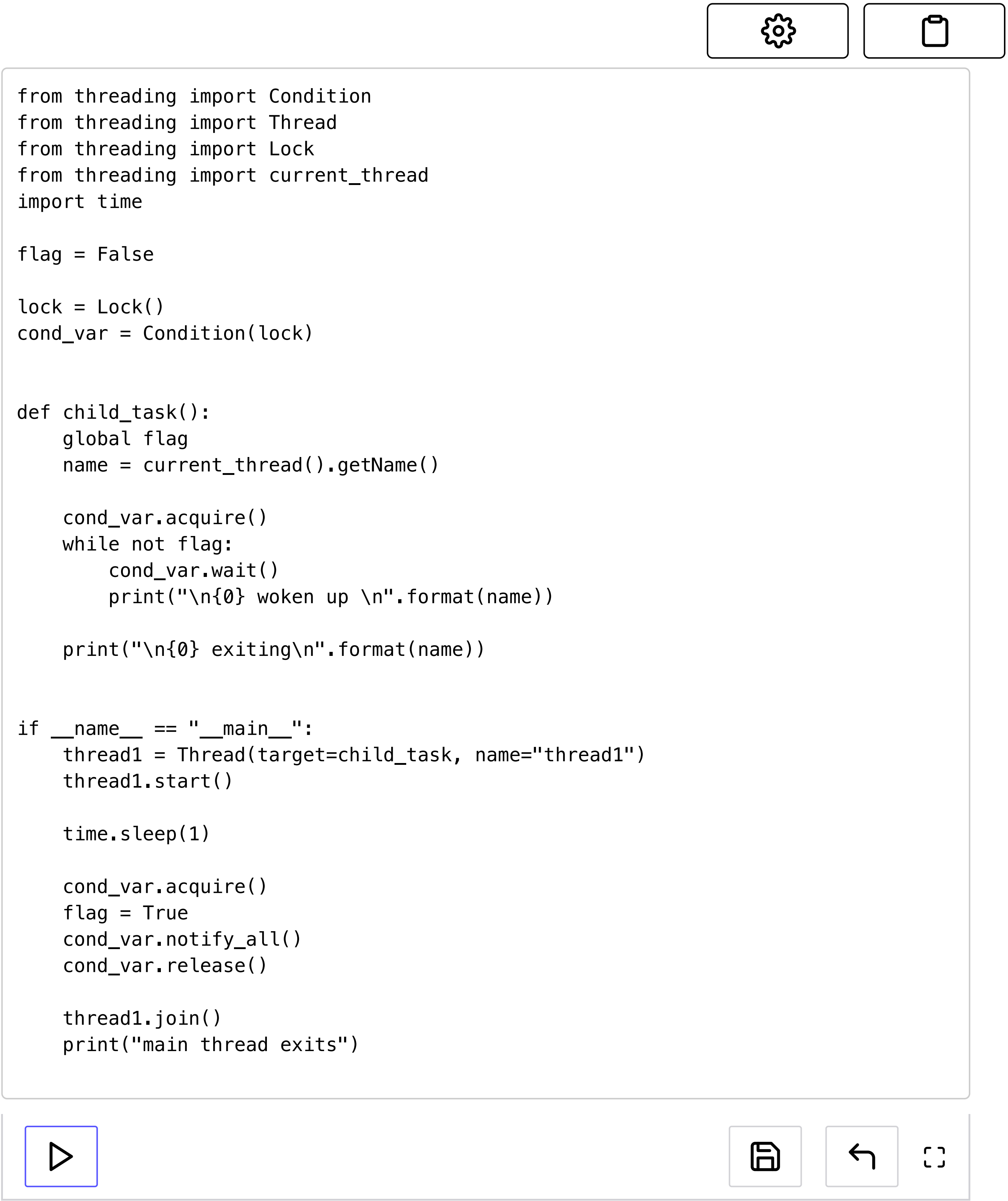
Q Does the program hang?

○ A) Yes

○ B) No

Submit Answer

Reset Quiz



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

flag = False

lock = Lock()
cond_var = Condition(lock)


def child_task():
    global flag
    name = current_thread().getName()

    cond_var.acquire()
    while not flag:
        cond_var.wait()
        print("\n{0} woken up \n".format(name))

    print("\n{0} exiting\n".format(name))


if __name__ == "__main__":
    thread1 = Thread(target=child_task, name="thread1")
    thread1.start()

    time.sleep(1)

    cond_var.acquire()
    flag = True
    cond_var.notify_all()
    cond_var.release()

    thread1.join()
    print("main thread exits")
```

Question

> ***Consider the code in the previous question gets an additional***

thread.

```
flag = False

lock = Lock()
cond_var = Condition(lock)


def child_task():
    global flag
    name = current_thread().getName()

    cond_var.acquire()
    while not flag:
        cond_var.wait()
        print("\n{0} woken up \n".format(name), flush=True)

    print("\n{0} exiting\n".format(name), flush=True)


if __name__ == "__main__":
    thread1 = Thread(target=child_task, name="thread1")
    thread1.start()

    thread2 = Thread(target=child_task, name="thread2")
    thread2.start()

    time.sleep(1)

    cond_var.acquire()
    flag = True
    cond_var.notify_all()
    cond_var.release()

    print("main thread exits", flush=True)
```

Q

The main thread now spawns two threads and the **`child_task`** is still missing the release lock statement. An intern on your team sees the code hanging and simply removes the join thread statements from the main thread's code. Will the program hang now?

○ **A)** Yes

○ **B)** No

Submit Answer

Reset Quiz

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

flag = False

lock = Lock()
cond_var = Condition(lock)


def child_task():
    global flag
    name = current_thread().getName()

    cond_var.acquire()
    while not flag:
        cond_var.wait()
        print("\n{0} woken up \n".format(name), flush=True)

    print("\n{0} exiting\n".format(name), flush=True)


if __name__ == "__main__":
    thread1 = Thread(target=child_task, name="thread1")
    thread1.start()

    thread2 = Thread(target=child_task, name="thread2")
    thread2.start()

    time.sleep(1)

    cond_var.acquire()
    flag = True
    cond_var.notify_all()
    cond_var.release()

    print("main thread exits", flush=True)
```

← Back

... continued

Next →

Semaphores

☑ Mark as Completed

Report an Issue

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