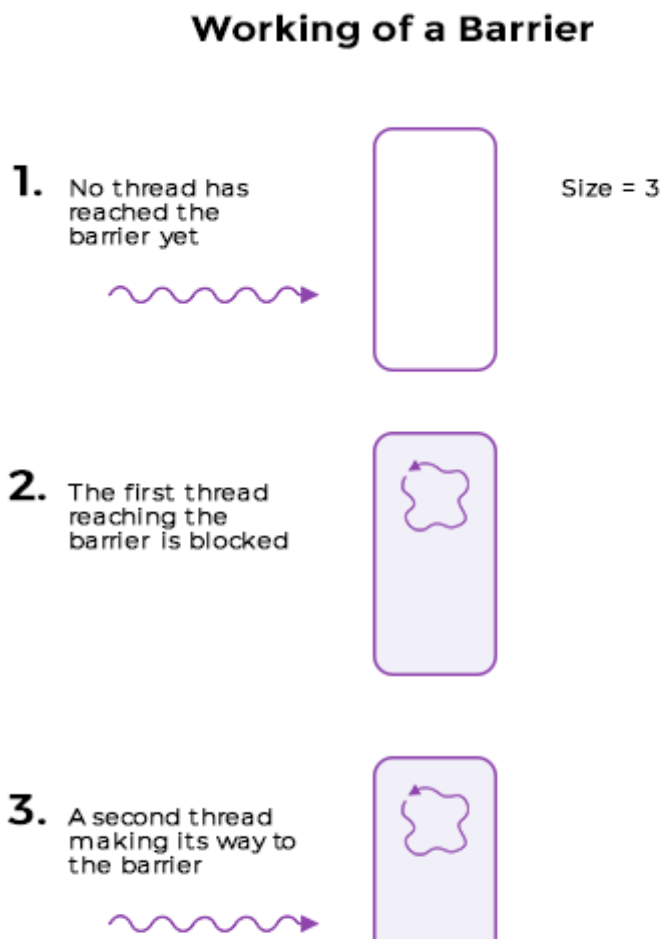
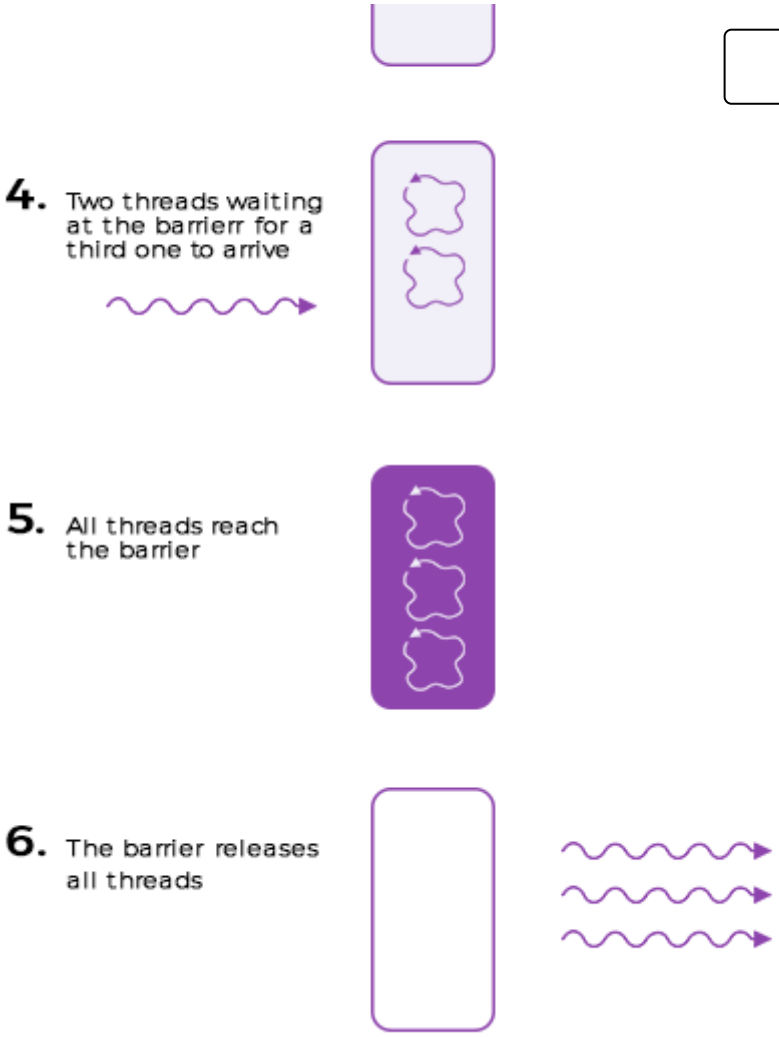
Implementing a Barrier

This lesson discusses how a barrier can be implemented in Python.

Implementing a Barrier

> A barrier can be thought of as a point in the program code, which all or some of the threads need to reach at before any one of them is allowed to proceed further.





Solution

A barrier allows multiple threads to congregate at a point in code before any one of the thread is allowed to move forward. Python and most other languages provide libraries which make barrier construct available for developer use. Even though we are re-inventing the wheel but this makes for a good interview question.

We can immediately realize that our solution will need a count variable to track the number of threads that have arrived at the barrier. If we have ***n*** threads, then ***n-1*** threads must wait for the **nth** thread to arrive. This suggests we have the **n-1** threads execute the wait method and the

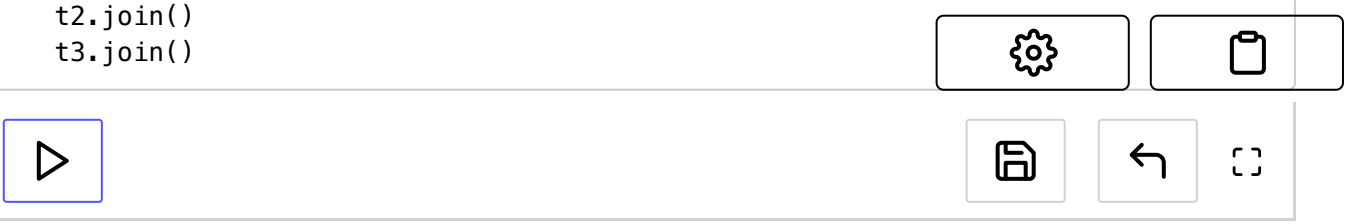
nth thread wakes up all the asleep **n-1** threads.

Below is the code:

```
class Barrier(object):
    def __init__(self, size):
        self.barrier_size = size
        self.reached_count = 0
        self.cond = Condition()

    def arrived(self):
        self.cond.acquire()
        self.reached_count += 1

        if self.reached_count == self.barrier_size:
            self.cond.notifyAll()
            self.reached_count = 0
        else:
            self.cond.wait()

        self.cond.release()
```

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


class Barrier(object):
    def __init__(self, size):
        self.barrier_size = size
        self.reached_count = 0
        self.cond = Condition()

    def arrived(self):
        self.cond.acquire()
        self.reached_count += 1

        if self.reached_count == self.barrier_size:
            self.cond.notifyAll()
            self.reached_count = 0
        else:
            self.cond.wait()

        self.cond.release()


def thread_process(sleep_for):
    time.sleep(sleep_for)
    print("Thread {0} reached the barrier".format(current_thread().getName()))
    barrier.arrived()

    time.sleep(sleep_for)
    print("Thread {0} reached the barrier".format(current_thread().getName()))
    barrier.arrived()

    time.sleep(sleep_for)
    print("Thread {0} reached the barrier".format(current_thread().getName()))
    barrier.arrived()


if __name__ == "__main__":
    barrier = Barrier(3)

    t1 = Thread(target=thread_process, args=(0,))
    t2 = Thread(target=thread_process, args=(0.5,))
    t3 = Thread(target=thread_process, args=(1.5,))

    t1.start()
    t2.start()
    t3.start()

    t1.join()
```

```
t2.join()
t3.join()
```



When you run the above code, you'll see that the threads print themselves in order, i.e. first thread 1 then thread 2 and finally thread 3. Thread 1 after reaching the barrier waits for the other two threads to reach the barrier before moving forward.

The above code has a subtle but very crucial bug! Can you spot the bug and try to fix it before reading on?

Second Cut

The previous code would have been hunky-dory if we were guaranteed that no spurious wake-ups could ever occur. The `wait()` method invocation without an enclosing while loop is an error. We discussed in previous sections that `wait()` should always be used with a while loop that checks for a condition, and if found false, should make the thread wait again.

The condition the while loop can check for is simply how many threads have incremented the `reached_count` variable so far. A thread that wakes up spuriously should go back to waiting if the `reached_count` is less than the size of the barrier. We can check for this condition as follows:

```
while self.reached_count < self.barrier_size
    wait();
```

The while loop introduces another problem. When the last thread does a `notifyAll()` it also resets the `reached_count` to 0, which means the threads that are legitimately woken up will always be stuck in the while

threads that are legitimately woken up will always be stuck in the while loop because `reached_count` is immediately set to zero. What we really want is not to reset the `reached_count` variable to zero until all the threads escape the while condition when `reached_count` becomes `barrier_size`. Below is the improved version:

```
0. class Barrier(object):
1.     def __init__(self, size):
2.         self.barrier_size = size
3.         self.reached_count = 0
4.         self.released_count = self.barrier_size
5.         self.cond = Condition()
6.
7.     def arrived(self):
8.
9.         self.cond.acquire()
10.
11.        self.reached_count += 1
12.
13.        if self.reached_count == self.barrier_size:
14.            self.released_count = self.barrier_size
15.
16.        else:
17.             while self.reached_count < self.barrier_size:
18.                self.cond.wait()
19.
20.        self.released_count -= 1
21.
22.        if self.released_count == 0:
23.            self.reached_count = 0
24.
25.        self.cond.notifyAll()
26.        self.cond.release()
```

The above code introduces a new variable `released_count` that keeps tracks of how many threads exit the barrier, and when the last thread exits the barrier it resets `reached_count` to zero, so that the barrier object can be reused in the future.

There is still a bug in the above code! Can you guess what it is?

Final Version

To understand why the above code is broken, consider three threads **t1, t2, and t3** trying to await on a barrier object in an infinite loop. Note the following sequence of events:

1. Threads t1 and t2 invoke `arrived()` and end up waiting at **line#18**. The `reached_count` variable is set to 2 and any spurious wakeups will cause t1 and t2 to go back to waiting. So far so good.

2. Threads t3 comes along, executes the if block on **line#13** and finds `reached_count == barrier_size` condition to be true. Thread t3 doesn't wait, notifies threads t1 and t2 to wake up, and exits.

3. **If thread t3 attempts to invoke `arrived()` immediately after exiting it and is successful before threads t1 or t2 get a chance to acquire the condition variable back, then the reached_count variable will be incremented to 4.**

4. With `reached_count` equal to 4, t3 will not block at the barrier and exit which breaks the contract for the barrier.

5. The invocation order of the `arrived()` method was t1, t2, t3, and then t3 again. The right behaviour would have been to release t1, t2, or t3 in any order and then block t3 on its second invocation of the `arrived()` method.

6. Another flaw with the above code is that it can cause a deadlock. Suppose we wanted the three threads t1, t2, and t3 to congregate at a barrier twice. The first invocation was in the order [t1, t2, t3] and the second was in the order [t3, t2, t1]. If t3 immediately invoked `arrived()` after the first barrier, it would go past the second

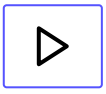
barrier without stopping while t2 and t1 would become stranded at the second barrier, since **`reached_count`** would never equal **`barrier_size`**.

The fix requires us to block any new threads from proceeding until all the threads that have reached the previous barrier are released. The code with the fix appears below:

```
class Barrier(object):
    def __init__(self, size):
        self.barrier_size = size
        self.reached_count = 0
        self.released_count = self.barrier_size
        self.cond = Condition()

    def arrived(self):

        self.cond.acquire()

        while self.reached_count == self.barrier_size:
            self.cond.wait()

        self.reached_count += 1

        if self.reached_count == self.barrier_size:
            self.released_count = self.barrier_size
        else:
            while self.reached_count < self.barrier_size:
                self.cond.wait()

        self.released_count -= 1

        if self.released_count == 0:
            self.reached_count = 0

        print("{0} released".format(current_thread().getName
()))
        self.cond.notifyAll()
        self.cond.release()
```

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


class Barrier(object):
    def __init__(self, size):
        self.barrier_size = size
        self.reached_count = 0
        self.released_count = self.barrier_size
        self.cond = Condition()

    def arrived(self):

        self.cond.acquire()

        while self.reached_count == self.barrier_size:
            self.cond.wait()

        self.reached_count += 1

        if self.reached_count == self.barrier_size:
            self.released_count = self.barrier_size
        else:
            while self.reached_count < self.barrier_size:
                self.cond.wait()

        self.released_count -= 1

        if self.released_count == 0:
            self.reached_count = 0

        print("{0} released".format(current_thread().getName()), flush=True)
        self.cond.notifyAll()
        self.cond.release()


def thread_process(sleep_for):
    time.sleep(sleep_for)
    print("Thread {0} reached the barrier".format(current_thread().getName()), flus
    barrier.arrived()

    time.sleep(sleep_for)
    print("Thread {0} reached the barrier".format(current_thread().getName()))
    barrier.arrived()

    time.sleep(sleep_for)
    print("Thread {0} reached the barrier".format(current_thread().getName()))
    barrier.arrived()
```

```
if __name__ == "__main__":

    barrier = Barrier(3)

    t1 = Thread(target=thread_process, args=(0,))
    t2 = Thread(target=thread_process, args=(0.5,))
    t3 = Thread(target=thread_process, args=(1.5,))

    t1.start()
    t2.start()
    t3.start()

    t1.join()
    t2.join()
    t3.join()
```




Back
Unisex Bathroom Problem

Next
Uber Ride Problem

Mark as Completed

Report an Issue

Ask a Question (https://discuss.educative.io/tag/implementing-a-barrier__interview-practise-problems__python-concurrency-for-senior-engineering-interviews)