... continued

Using a Background Thread

The previous solution consisted of manipulating pointers in time, thus avoiding threads altogether. Another solution is to use threads to solve the token bucket filter problem. We instantiate one thread to add a token to the bucket after every one second. The user thread invokes the **`getToken()`** method and is granted one if available. We instantiate one thread to add a token to the bucket after every one second. The user thread invokes the **`getToken()`** method and is granted one if available.

One simplification as a result of using threads is we now only need to remember the current number of tokens held by the token bucket filter object. We'll add an additional method **`daemonThread()`** that will be executed by the thread that adds a token every second to the bucket. The skeleton of our class looks as follows:

```
class MultithreadedTokenBucketFilter(TokenBucketFilter):
    def __init__(self, maxTokens):
        self.MAX_TOKENS = int(maxTokens)
        self.possibleTokens  = int(0)
        self.ONE_SECOND = int(1)
        self.cond = Condition()

    def daemonThread(self):

    def getToken(self):
```

The logic of the daemon thread is simple. It sleeps for one second, wakes up, checks if the number of tokens in the bucket is less than the maximum allowed tokens, if yes increments the **possibleTokens** variable and if not goes back to sleep for a second again.

The implementation of the **getToken()** is even simpler. The user thread checks if the number of tokens is greater than zero, if yes it simulates taking away a token by decrementing the variable **possibleTokens**. If the number of available tokens is zero then the user thread must wait and be notified only when the daemon thread has added a token. The implementation of the **getToken()** method is shown below:

```
    def getToken(self):
        self.cond.acquire()
        while self.possibleTokens == 0:
            self.cond.wait()
        self.possibleTokens = self.possibleTokens-1;
        self.cond.release();

        print("Granting " + threading.current_thread().getName
() + " token at " + str(datetime.now()));
```

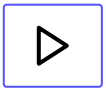
Note that we are manipulating the shared mutable variable **possibleTokens** in a critical section. Additionally, we **wait()** when the number of tokens is zero. The implementation of the daemon thread is given below. It runs in a perpetual loop.

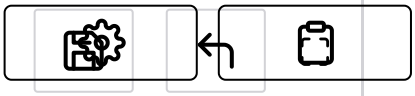
```
    def daemonThread(self):
        while True:
            self.cond.acquire()
            if self.possibleTokens < self.MAX_TOKENS:
                self.possibleTokens = self.possibleTokens + 1;
            self.cond.notify()
            self.cond.release()
```

```
            time.sleep(self.ONE_SECOND);
```

The complete implementation along with a test-case appears in the code widget below:

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

class MultithreadedTokenBucketFilter():
    def __init__(self, maxTokens):
        self.MAX_TOKENS = int(maxTokens)
        self.possibleTokens  = int(0)
        self.ONE_SECOND = int(1)
        self.cond = Condition()
        dt = Thread(target = self.daemonThread);
        dt.setDaemon(True);
        dt.start();


    def daemonThread(self):
        while True:
            self.cond.acquire()
            if self.possibleTokens < self.MAX_TOKENS:
                self.possibleTokens = self.possibleTokens + 1;
            self.cond.notify()
            self.cond.release()

            time.sleep(self.ONE_SECOND);

    def getToken(self):
        self.cond.acquire()
        while self.possibleTokens == 0:
            self.cond.wait()
        self.possibleTokens = self.possibleTokens - 1
        self.cond.release()

        print("Granting " + current_thread().getName() + " token at " + str(time.ti

if __name__ == "__main__":
    threads_list = [];
    bucket = MultithreadedTokenBucketFilter(1)

    for x in range(10):
        workerthread =  Thread(target=bucket.getToken)
        workerthread.name = "Thread_" + str(x+1)
        threads_list.append(workerthread)


    for t in threads_list:
        t.start()

    for t in threads_list:
        t.join()
```





We reuse the test-case from the previous lesson, where we create a token bucket with max tokens set to 1 and have ten threads request for a token. The threads are shown being granted tokens at exactly 1-second intervals instead of all at once. The program output displays the timestamps at which each thread gets the token and we can verify the timestamps are 1 second apart. Additionally, we mark the daemon thread as background so that it exits when the application terminates. Additionally, we mark the daemon thread as background so that it exits when the application terminates.

Using a Factory

The problem with the above solution is that we start our thread in the constructor. **Generally, it is inadvisable to start a thread in a constructor as the child thread can be passed the `self` variable which the child thread can use before the object pointed to by the passed-in `self` is fully constructed.**

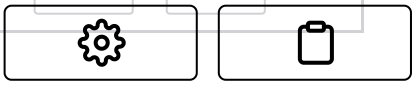
There are two ways to overcome this problem, the naive but correct solution is to start the daemon thread outside of the **`MultithreadedTokenBucketFilter`** object. However, the con of this approach is that the management of the daemon thread spills outside the class. Ideally, we want the class to encapsulate all the operations related with the management of the token bucket filter and only expose the public API to the consumers of our class, as per good object orientated design. This situation is a great for using the **Simple Factory** design pattern. We'll create a factory class which produces token bucket filter objects and also starts the daemon thread only when the object is full constructed.

The complete code with the same test case appears below.

main.py

TokenBucketFilterFactory.py

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

class TokenBucketFilterFactory:

    @staticmethod
    def makeTokenBucketFilter(capacity):
        tbf = MultithreadedTokenBucketFilter(capacity
        tbf.initialize();
        return tbf;


class MultithreadedTokenBucketFilter:
    def __init__(self, maxTokens):
        self.MAX_TOKENS = int(maxTokens)
        self.possibleTokens  = int(0)
        self.ONE_SECOND = int(1)
        self.cond = Condition()

    def initialize(self):
        dt = Thread(target = self.daemonThread);
        dt.setDaemon(True);
        dt.start();

    def daemonThread(self):
        while True:
            self.cond.acquire()
            if self.possibleTokens < self.MAX_TOKENS:
                self.possibleTokens = self.possibleTo
            self.cond.notify()
            self.cond.release()

            time.sleep(self.ONE_SECOND);

    def getToken(self):
        self.cond.acquire()
        while self.possibleTokens == 0:
            self.cond.wait()
        self.possibleTokens = self.possibleTokens - 1
        self.cond.release()

        print("Granting " + current_thread().getName(
```



Back

Next

Rate Limiting Using Token Bucket Filter

Thread Safe Deferred Callback

Mark as Completed

Report an Issue

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