Thread Safety

This lessons discusses the concept of thread safety.

Thread Safety

The primary motivation behind using multiple threads is improving program performance that may be measured with metrics such as throughput, responsiveness, latency, etc. Whenever threads are introduced in a program, the shared state amongst the threads becomes vulnerable to corruption. If a class or a program has immutable state then the class is necessarily thread-safe. Similarly, the shared state in an application where the same thread mutates the state using an operation that translates into an atomic bytecode instruction can be safely read by multiple reader threads. In contrast, a sole writer thread mutating the shared state using several atomic bytecode instructions isn't a thread-safe scenario for reader threads. Most multi-threaded setups require caution when interacting with shared state. As a corollary, the composition of two thread-safe classes doesn't guarantee thread-safety.

Atomicity

Consider the below snippet:

```
count = 0

def increment():
    global count
    count += 1
```

The above code will work flawlessly when it is executed by a single thread. However, if there are two or more threads involved, things get tricky. The key to realize is that the statement `count += 1` isn't atomic. A thread can't increment the variable atomically, i.e. there doesn't exist a single bytecode instruction that can increment the count variable. Let's examine the bytecode generated for our snippet above.

```
import dis

count = 0

def increment():
    global count
    count += 1

# prints the bytecode
dis.dis(increment)

```

Output 0.56s

```
  7           0 LOAD_GLOBAL              0 (count)
              3 LOAD_CONST               1 (1)
              6 INPLACE_ADD
              7 STORE_GLOBAL             0 (count)
             10 LOAD_CONST               0 (None)
             13 RETURN_VALUE
```

Generated Byte Code

7 0 LOAD_GLOBAL 0 (count)
3 LOAD_CONST 1 (1)
6 INPLACE_ADD
7 STORE_GLOBAL 0 (count)
10 LOAD_CONST 0 (None)
13 RETURN_VALUE

The seemingly single line statement expands into multiple bytecode instructions. When two threads invoke the `increment()` method it is possible that the first thread is switched out by the Python interpreter just before the third **INPLACE_ADD** instruction is executed. Now the second thread comes along and executes all the six bytecode instructions in one go. When the first thread is rescheduled by the interpreter, it executes the third line but the value the thread holds is stale causing it to incorrectly update the `count` variable.

Programming languages provide constructs such as mutexes and locks to help developers guard sections of code that must be executed sequentially by multiple threads. Guarding shared data is one aspect of multi-threaded programs. The other aspect is coordination and cooperation amongst threads. Again, languages provide mechanisms to facilitate threads to work cooperatively towards a common goal. These include semaphores, barriers etc.

Thread Unsafe Class

Take a minute to go through the following program. It increments an object of class `Counter` using 5 threads. Each thread increments the object a hundred thousand times. The final value of the counter should be half a million (500,000). If you run the program enough times, you'll sometimes get the correct summation, and at others, you'll get an incorrect value.

```
from threading import Thread
import sys

class Counter:

```

```python
    def __init__(self):
        self.count = 0
```



```python
            self.count += 1


if __name__ == "__main__":

    # Sets the thread switch interval
    sys.setswitchinterval(0.005)

    numThreads = 5
    threads = [0] * numThreads
    counter = Counter()

    for i in range(0, numThreads):
        threads[i] = Thread(target=counter.increment)

    for i in range(0, numThreads):
        threads[i].start()

    for i in range(0, numThreads):
        threads[i].join()
```


Output 0.63s

```
count = 352061
```

Fixing Thread Unsafe Class

We fix the above example using the equivalent of a mutex in Python called a **Lock**. For now, don't worry about how the example below works, but observe how the count always sums up to half a million.

```python
from threading import Thread
from threading import Lock
import sys

class Counter:

    def __init__(self):
        self.count = 0
        self.lock = Lock()

    def increment(self):
        for _ in range(100000):
            self.lock.acquire()
            self.count += 1
            self.lock.release()


if __name__ == "__main__":
```

```
    # Sets the thread switch interval
    sys.setswitchinterval(0.005)
```

Select lesson text to explore the Explain with AI Premium Offering

```
    counter = Counter()

    for i in range(0, numThreads):
        threads[i] = Thread(target=counter.increment)

    for i in range(0, numThreads):
```

Run

Save Reset

Output 2.1s

```
count = 500000
```


Back

I/O Bound vs CPU Bound

Mark As Completed

Next

Critical Section & Race Conditions