PiterPy

Tackling Thread Safety in Python

Piterpy 2024

DIGIEVO LABS

About Me

Adarsh Divakaran

Co-founder and Lead consultant

Digievo Labs

DIGIEVO LABS

PiterPy

Outline

- Threading
- Race Conditions
- Making Programs thread safe using **Synchronization primitives**
    - Lock
    - RLock
    - Semaphore
    - Event
    - Barrier
    - Condition

PiterPy

Threading

Python’s **threading** is a concurrency framework that allows us to spin up multiple threads that can run concurrently, each executing pieces of code, improving the efficiency and responsiveness of our application.

Why use threading?

- To improve application efficiency (concurrent execution, improve responsiveness)

DIGIEVO LABS

Sample - A Travel Booking Application

```
available_seats = 10


def book_seat():
    global available_seats
    if available_seats > 0:
        time.sleep(0.1) # Simulate processing
        available_seats -= 1
        print(f"Seat booked. Remaining seats: {available_seats}")
    else:
        print("Sorry, no seats available.")
```

Sample - A Travel Booking Application

Problem: Bookings get completed sequentially - slower

Only one user can access the Python script simultaneously

Solution: Use multithreading

Multi-Threaded Booking App

```
concurrent_threads = 10


with concurrent.futures.ThreadPoolExecutor(max_workers=concurrent_threads) as executor:
    for i in range(15):
        executor.submit(book_seat)
```

PiterPy

Demo

DIGIEVO LABS

Multi-Threaded Booking App

Debugging the issue:

- **Concurrent** read & write to shared data happens
- This can lead to **race conditions**

Race Conditions

A **race condition** occurs when the outcome of a program depends on the sequence or timing of uncontrollable events like thread execution order.

- Race conditions occur when we use threading with **shared mutable data**.
- Non atomic operations can get **context switched** in between.

Race Conditions - Context Switching

PiterPy

Demo

DIGIEVO LABS

Thread Safety

A program is said to be **thread-safe** if it can be run using multiple threads **without** any unexpected **side effects** (or race conditions)

When should we worry about Thread safety

- We are using threading as our concurrency framework

Parallel execution in Python

Model	Execution	Start-up time	Data Exchange	Best for...
threads	Parallel *	**small**	Any	Small, IO-bound tasks that don't require multiple CPU cores
coroutines	Concurrent	**smallest**	Any	
multiprocessing	Parallel	**large**	Serialization	Larger, CPU or IO-bound tasks that require multiple CPU cores
Sub Interpreters	Parallel	**medium****	Serialization or Shared Memory	

[Ref: Anthony Shaw - Unlocking the Parallel Universe: Subinterpreters and Free-Threading in Python 3.13 - Pycon US 2024]

When should we worry about Thread safety

Code contains **shared mutable data** & **non-atomic operations**

- Threads share memory location of parent process
- No problem if no data is shared
- No problem if code executed with threads operates on immutable data and the operations are atomic

Making programs thread safe

Options:

- Don’t use threads (go with other concurrency frameworks)
- Don’t share mutable data across threads - Use thread local data
- Use Synchronization primitives.
- Make operations atomic (Python bytecode level)

Synchronization Primitives

- Lock
- RLock
- Semaphore
- Condition
- Event
- Barrier

Synchronization Primitives - Lock

A Lock is a synchronization primitive that allows only one thread to access a resource at a time.

Practical Use-Case: Ensuring that only one thread can modify a shared variable at a time to prevent race conditions.

Synchronization Primitives - Lock

```
lock = threading.Lock()


def book_seat():
    lock.acquire() # other threads get blocked (waiting for lock release) here
    try:
        # only one thread can access this critical section at a time
        pass
    finally:
        lock.release()
```

Synchronization Primitives - Lock

```python
lock = threading.Lock()


def book_seat():
    # the critical section (enclosed in the `with block`) is protected by the lock
    with lock:
        # only one thread can access this critical section at a time
        pass
```

DIGIEVO LABS

Demo

- Seat booking application
- 10 worker threads, 15 workers try to book at the same time
- A Lock object is used to mark the critical section of code

Synchronization Primitives - RLock

An RLock is a reentrant lock that allows the same thread to acquire the lock multiple times without causing a deadlock.

Practical Use-Case: Allowing a thread to re-enter a critical section of code that it already holds the lock for, such as in recursive functions.

Usage is same as that of Lock (acquire and release methods and context manager).

Demo

- Example uses only 1 thread
- We have a function which acquires a lock and calls another function
- The called function also need to acquire the lock

Synchronization Primitives - Semaphore

A Semaphore is a synchronization primitive that controls access to a resource by maintaining a counter, allowing a set number of threads to access the resource simultaneously.

Practical Use-Case: Limiting the number of concurrent connections to a database to prevent overload. (eg: connection pooling)

Synchronization Primitives - Semaphore

```python
max_concurrent_bookings = 3
semaphore = threading.Semaphore(max_concurrent_bookings)


def book_travel_package():

    # any number of threads can enter up to here
    with semaphore:
        # only 3 threads can enter this code block simultaneously
        # other threads should wait
        ...
```

Demo

- 10 Workers trying to book a travel package
- We allow only 3 concurrent bookings using a semaphore
- All threads start at the same time, but only 3 threads can perform booking at a time.

Synchronisation Primitives - Event

An Event is a synchronization primitive that allows one thread to signal one or more other threads that a particular condition has been met.

Practical Use-Case: Notifying worker threads that new data is available for processing.

Synchronization Primitives - Event

```python
flight_landed = threading.Event()


def wait_for_passengers():
    # wait for the event to be set.
    flight_landed.wait()
    # code to be executed after the event happened


def flight_status_update():
    # perform some operations
    flight_landed.set()
```

Demo

- We have 2 threads
- 1 - To update flight status after a delay
- 2 - To collect passengers
- Both start at the same time
- They use an event object for communication

Synchronization Primitives - Barrier

A Barrier is a synchronization primitive that allows multiple threads to wait until all threads have reached a certain point before any of them can proceed.

Practical Use-Case: Ensuring that all worker threads complete their individual tasks before any thread proceeds to the next phase of a multi-phase computation.

Synchronization Primitives - Barrier

```
num_travelers = 4
barrier = threading.Barrier(num_travelers + 1)  # +1 for the tour guide


def traveler():
    # code to get the traveller ready
    barrier.wait()
    # this line gets executed only when all travelers (total 5) are ready


def tour_guide():
    # independent operations
    barrier.wait()
    # code here will execute after 5 threads are waiting at the barrier
```

Demo

- Example where there are 4 travellers and a tour guide
- Guide is ready from the start
- Travellers take a random time to get ready
- Using a barrier, the tour starts at the time when all of them becomes ready

Synchronization Primitives - Condition

A Condition is a synchronization primitive that allows threads to wait for certain conditions to be met before continuing execution. It is associated with a lock which can be used for exclusive access to any resource once the condition is met.

Practical Use-Case: Pausing a thread until a specific condition is met, such as waiting for a queue to be non-empty before consuming an item (producer-consumer scenarios).

Synchronization Primitives - Condition

```
customer_available_condition = threading.Condition()

customer_queue = []


def add_customer_to_queue(customer_name):

    with customer_available_condition:

        customer_queue.append(customer_name)

        customer_available_condition.notify()
```

Synchronization Primitives - Condition

```python
customer_available_condition = threading.Condition()
customer_queue = []


def serve_customers():
    while True:
        with customer_available_condition:
            # Wait for a customer to arrive
            while not customer_queue:
                customer_available_condition.wait() # Blocks here unless notified

            customer = customer_queue.pop(0) # Get and serve the customer
```

Synchronization Primitives - Condition

A condition object is always associated with some kind of lock; this can be passed in or one will be created by default. It has the below methods:

- The `wait()` method releases the lock, and then blocks until another thread awakens it by calling notify methods. Once awakened, `wait()` re-acquires the lock and returns. It is also possible to specify a timeout.
- The `notify()` method wakes up one of the threads waiting for the condition variable, if any are waiting. The `notify_all()` method wakes up all threads waiting for the condition variable.

Demo

- We have 1 travel agent and 5 customers
- The travel agent serves customers one at a time
- Customers are stored in `customer_queue`
- The queue is shared across threads, so we use the condition object as a lock.
- The `customers` list stores the name of the customers and the delay of their arrival.
- Customer gets served immediately when the travel agent is free.
- If the travel agent is busy, they should wait until the current customer is done being served.

Condition - Difference from Other Primitives

A condition object is always associated with some kind of lock; this can be passed in or one will be created by default. It has the below methods:

- A condition involves a **lock + additional methods**
- We can use the condition object as a lock

The **lock** will be used when we use the condition object as a context manager.

- Whenever the **.wait()** method is called, it releases the lock.

Condition vs Event

- Event objects are commonly used to handle **one-time** events.
- Conditions are used for producers-consumer scenarios.
- Conditions are suited when there is a **continuous flow** of events happening.
- Condition = Handle events + Get exclusive access (using lock)

Summary

- Before moving to **multithreading** keep in mind that the code you are working with might not be designed for thread safety - even library code.
- Before switching to multithreading, check for **shared mutable data & atomicity** requirements.
- Add **synchronization primitives** to enforce thread-safety.

“When in doubt, use a mutex!” - CPython docs

(https://docs.python.org/3/faq/library.html#what-kinds-of-global-value-mutation-are-thread-safe)

Thank You

Get the talk materials & connect with me

linkhq.co/adarsh

PiterPy

DIGIEVO LABS