Tackling Thread Safety in Python

Europython 2024

Speakers

Adarsh Divakaran

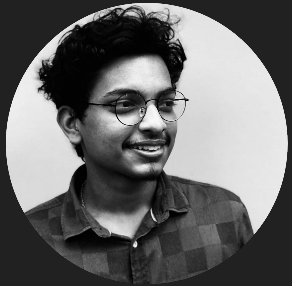
Co-founder and Lead consultant

Digievo Labs

Jothir Adithyan

Product Engineer

Strollby, UST



Outline

- Threading
- Race Conditions
- Thread Safety
- Synchronization primitives
- Making Programs thread safe

Threading

Why use threading?

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

Sample - A banking app

Banking App - DB Init

```python
from sqlmodel import Field, Session, SQLModel, create_engine


def setup_db(engine):
    with Session(engine) as session:
        statement = delete(Account)
        result = session.exec(statement)
        session.commit()

    acc_1 = Account(id=1, name="John", balance=100)
    acc_2 = Account(id=2, name="Jane", balance=100)
    acc_3 = Account(id=3, name="Alice", balance=100)

    with Session(engine) as session:
        session.add(acc_1)
        session.add(acc_2)
        session.add(acc_3)
        session.commit()


class Account(SQLModel, table=True):
    id: int = Field(primary_key=True)
    name: str
    balance: float
```

Banking App - Initial Balance

id	name	balance
1	John	100
2	Jane	100
3	Alice	100

Money in bank - 300

Banking App - Transfer Code

```python
from sqlmodel import Session, select

from db import engine, Account


def transfer_money(from_id, to_id, amount):
    with Session(engine) as session:
        from_user = select(Account).where(Account.id == from_id)
        to_user = select(Account).where(Account.id == to_id)
        from_user = session.exec(from_user).one()
        to_user = session.exec(to_user).one()
        from_user.balance -= amount
        to_user.balance += amount
        session.add(from_user)
        session.add(to_user)
        session.commit()


def display_balance():
    with Session(engine) as session:
        statement = select(Account)
        results = session.exec(statement)
        for result in results:
            print(result.balance)
```

Banking App - Money Transfer (Single-Thread)

```python
if __name__ == '__main__':

    init_db()
    display_balance()

    print("doing transfers")
    to_transfer = [(1, 2, 10), (2, 3, 10), (1, 3, 10), (3, 1, 10)]
    for from_id, to_id, amount in to_transfer:
        transfer_money(from_id, to_id, amount)

    display_balance()
```

Banking App - Balance After Transfer

id	name	balance
1	John	90
2	Jane	100
3	Alice	110

Money in bank - 300

Banking App - Performance Issues

Transfers get completed sequentially - slower

Banking App - Money Transfer (Multi-Thread)

```python
if __name__ == '__main__':

    init_db()
    display_balance()

    print("doing transfers")
    to_transfer = [(1, 2, 10), (2, 3, 10), (1, 3, 10), (3, 1, 10)]

    with concurrent.futures.ThreadPoolExecutor(max_workers=10) as executor:
        for from_id, to_id, amount in to_transfer:
            executor.submit(transfer_money, from_id, to_id, amount)

    display_balance()
```

Banking App - Balance After Transfer

id	name	balance
1	John	90
2	Jane	90
3	Alice	110

Money in bank - 290

Banking App - Debugging the Issue

Concurrent read & write happens due to threading

This can lead to race conditions

Race Conditions

Race conditions occur when we work with shared mutable data

Non atomic operations can get context switched in between

Race Conditions - Context Switching

```python
import sys
import threading
import time


def worker(id_):
    for i in range(3):
        time.sleep(0.05)
        print(f"Worker thread id {id_}; iteration {i}")


if __name__ == "__main__":
    print(f"{sys.getswitchinterval()=}")

    for i in range(3):
        threading.Thread(target=worker, args=(i,)).start()

"""
sys.getswitchinterval()=0.005
Worker thread id 2; iteration 0
Worker thread id 1; iteration 0
Worker thread id 0; iteration 0
Worker thread id 0; iteration 1
Worker thread id 1; iteration 1
Worker thread id 2; iteration 1
Worker thread id 2; iteration 2
Worker thread id 1; iteration 2
Worker thread id 0; iteration 2
"""
```

Race Conditions

Consider two threads in our banking app. A user has an initial balance of 30. An amount of 100 is being transferred simultaneously to the user by each of the thread.

1. Thread 1 reads the current balance (30)
2. Thread 1 updates the current balance (130)
3. Before thread 1 saves to the database, thread 2 reads the value of A (gets 30)
4. Thread 2 updates balance as 130.
5. Thread 2 writes value of 130 to DB; thread 1 also does the same.

The problem is that, a read is allowed midway of another modify operation.

Thread Safety

A program is said to be thread-safe if it can be run using multiple threads without any unexpected side effects (like the one we seen in banking example)

When should we worry about Thread safety

Is 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

Has shared mutable data & has non-atomic operations

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

Non thread-safe examples - Print

```python
import threading

def printer():
    message = f"Printing from thread: {threading.get_ident()}"
    end_separator = f" | Separator of {threading.get_ident()}\n"  # default end is '\n'
    print(message, end=end_separator)


for _ in range(5):
    printer()

"""
Printing from thread: 74112 | Separator of 74112
Printing from thread: 74112 | Separator of 74112
Printing from thread: 74112 | Separator of 74112
Printing from thread: 74112 | Separator of 74112
Printing from thread: 74112 | Separator of 74112
"""
```

Non thread safe examples - Print

- Print function operation - prints the value, Then prints separator and end (by default `\n`)
- It is thread unsafe because the operation is non atomic
- Context switch can happen in between

Non thread safe examples - Print

```python
from concurrent import futures
import threading


def printer():
    message = f"Printing from thread: {threading.get_ident()}"
    end_separator = f" | Separator of {threading.get_ident()}\n"  # default end is '\n'
    print(message, end=end_separator)


with futures.ThreadPoolExecutor(max_workers=3) as executor:
    for _ in range(8):
        executor.submit(printer)

"""
Printing from thread: 79484 | Separator of 79484
Printing from thread: 79484 | Separator of 79484
Printing from thread: 79484Printing from thread: 72360 | Separator of 72360
 | Separator of 79484
Printing from thread: 79484 | Separator of 79484
Printing from thread: 72360Printing from thread: 79484 | Separator of 79484
Printing from thread: 79484 | Separator of 79484
 | Separator of 72360
"""
```

Non thread safe examples - Singleton

```
class SingletonClass:
    def __new__(cls):
        if not hasattr(cls, 'instance'):
            cls.instance = super(SingletonClass, cls).__new__(cls)
        return cls.instance


obj1 = SingletonClass()
obj2 = SingletonClass()

print(obj1 is obj2) # True
print(id(obj1)) # 2402721138768
print(id(obj2)) # 2402721138768
```

Making programs thread safe

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

Synchronization Primitives

- Lock
- RLock
- Semaphore
- Event
- Condition
- 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 - 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.

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)

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 - Condition

A Condition is a synchronization primitive that allows threads to wait for certain conditions to be met before continuing execution.

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.

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.

Synchronisation Primitives - Lock

```python
import threading
import time

lock = threading.Lock()


def thread_func():
    print(f"Thread {threading.current_thread().ident} waiting to acquire lock")
    lock.acquire()
    try:
        # Critical section of code
        print(
            f"Lock acquired by {threading.current_thread().ident} at timestamp:{int(time.time())}, executing
critical section")
        time.sleep(5)
    finally:
        print(f"Lock Releasing by {threading.current_thread().ident}")
        lock.release()


thread1 = threading.Thread(target=thread_func).start()
thread2 = threading.Thread(target=thread_func).start()

"""
Thread 11848 waiting to acquire lock
Lock acquired by 11848 at timestamp:1720777564, executing critical section
Thread 29480 waiting to acquire lock
Lock Releasing by 11848
Lock acquired by 29480 at timestamp:1720777569, executing critical section
Lock Releasing by 29480
"""
```

Synchronisation Primitives - Lock

```python
import threading
import time

lock = threading.Lock()


def thread_func():
    print(f"Thread {threading.current_thread().ident} reached thread_func")
    with lock:
        # Critical section of code
        print(f"Lock acquired at {int(time.time())}, executing critical section")
        time.sleep(5)
        print(f"Lock Releasing by {threading.current_thread().ident}")


thread1 = threading.Thread(target=thread_func).start()
thread2 = threading.Thread(target=thread_func).start()
```

Synchronisation Primitives - Deadlock

```python
import threading
import time


class BankAccount:
    def __init__(self):
        self.balance = 0
        self.lock = threading.Lock()

    def deposit(self, amount):
        print(f"Thread {threading.current_thread().ident} waiting to acquire lock for deposit()")
        with self.lock:
            print(f"Thread {threading.current_thread().ident} acquired lock for deposit()")
            time.sleep(0.1)
            self._update_balance(amount)

    def _update_balance(self, amount):
        print(f"Thread {threading.current_thread().ident} waiting to acquire lock for _update_balance()")
        with self.lock:  # This will cause a deadlock
            print(f"Thread {threading.current_thread().ident} acquired lock for _update_balance()")
            self.balance += amount


account = BankAccount()


def make_deposits():
    account.deposit(100)


# Threads to perform deposits
threads = [threading.Thread(target=make_deposits) for _ in range(3)]
```

Synchronisation Primitives - Deadlock

```
Thread 27488 waiting to acquire lock for deposit()
Thread 27488 acquired lock for deposit()
Thread 6668 waiting to acquire lock for deposit()
Thread 22688 waiting to acquire lock for deposit()
Thread 27488 waiting to acquire lock for _update_balance()

(infinitely waiting...)
```

Synchronisation Primitives - RLock

```python
class BankAccount:
    def __init__(self):
        self.balance = 0
        self.lock = threading.RLock()

    def deposit(self, amount):
        print(f"Thread {threading.current_thread().ident} waiting to acquire lock for deposit()")
        with self.lock:
            print(f"Thread {threading.current_thread().ident} acquired lock for deposit()")
            time.sleep(0.1)
            self._update_balance(amount)

    def _update_balance(self, amount):
        print(f"Thread {threading.current_thread().ident} waiting to acquire lock for _update_balance()")
        with self.lock:  # Deadlock won't occur because or RLock usage
            print(f"Thread {threading.current_thread().ident} acquired lock for _update_balance()")
            self.balance += amount


account = BankAccount()


def make_deposits():
    account.deposit(100)


# Threads to perform deposits
threads = [threading.Thread(target=make_deposits) for _ in range(3)]
```

Synchronisation Primitives - RLock

```
Thread 23128 waiting to acquire lock for deposit()
Thread 23128 acquired lock for deposit()
Thread 11944 waiting to acquire lock for deposit()
Thread 27752 waiting to acquire lock for deposit()
Thread 23128 waiting to acquire lock for _update_balance()
Thread 23128 acquired lock for _update_balance()
Thread 11944 acquired lock for deposit()
Thread 11944 waiting to acquire lock for _update_balance()
Thread 11944 acquired lock for _update_balance()
Thread 27752 acquired lock for deposit()
Thread 27752 waiting to acquire lock for _update_balance()
Thread 27752 acquired lock for _update_balance()
Final balance: 300
```

Making Programs Thread Safe - Banking App

```python
account_lock = Lock()


def transfer_money(from_id, to_id, amount):
    with Session(engine) as session:
        # building SQL queries
        from_user = select(Account).where(Account.id == from_id)
        to_user = select(Account).where(Account.id == to_id)

        with account_lock:
            # executing SQL queries
            from_user = session.exec(from_user).one()
            to_user = session.exec(to_user).one()

            # Update the balance
            from_user.balance -= amount
            to_user.balance += amount

            # Save updated balance to DB
            session.add(from_user)
            session.add(to_user)
            session.commit()
```

Making Programs Thread Safe - Banking App

- The example here (using Python locks) is not suitable for production.
- Multiple instances of our Python app can be deployed across regions.
- Enable locks at the source of truth level.
- Enable locks at the database level

```python
def add_to_balance(account_number: str, transfer_amount: float):
    with Session(engine) as session:

        # Acquire a row-level lock
        statement = select(BankAccount).where(BankAccount.account_number == account_number).with_for_update()

        result = session.exec(statement).one_or_none()
        result.balance += transfer_amount
        session.add(result)
        session.commit()
```

Making Programs Thread Safe - Print

```python
from concurrent import futures
import threading

# Lock object
print_lock = threading.Lock()


def printer():
    message = f"Printing from thread: {threading.get_ident()}"
    end_separator = f" | Separator of {threading.get_ident()}\n"  # default end is '\n'

    # Acquire the lock before printing
    with print_lock:
        print(message, end=end_separator)


# Use ThreadPoolExecutor to run the printer function in multiple threads
with futures.ThreadPoolExecutor(max_workers=3) as executor:
    for _ in range(8):
        executor.submit(printer)

"""
Printing from thread: 13048 | Separator of 13048
Printing from thread: 13048 | Separator of 13048
Printing from thread: 13048 | Separator of 13048
Printing from thread: 13048 | Separator of 13048
Printing from thread: 3528 | Separator of 3528
Printing from thread: 13048 | Separator of 13048
Printing from thread: 3528 | Separator of 3528
Printing from thread: 3796 | Separator of 3796
"""
```

Making Programs Thread Safe - Singleton

```python
import threading


class Singleton:
  _instance = None
  _lock = threading.Lock()

  def __new__(cls):
      if cls._instance is None:
        with cls._lock:
          if cls._instance is None:
              cls._instance = super().__new__(cls)

      return cls._instance
```

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 slides & connect with us

linkhq.co/adarsh

linkhq.co/adithyan