Intro to Synchronisation

Synchronisation

> Synchronisation is the coordination of threads to ensure that they do not interfere with each other.

Whenever there are multiple threads that access the same resource, we need to make sure that the threads do not interfere with each other. This is called synchronisation.

Synchronisation can be seen in the adder and subtractor example. The adder and subtractor threads access the same counter variable. If the adder and subtractor threads do not synchronise, the counter variable can be in an inconsistent state.

- Create a count class that has a count variable.
- Create two different classes `Adder` and `Subtractor`.
- Accept a count object in the constructor of both the classes.
- In `Adder`, iterate from 1 to 100 and increment the count variable by 1 on each iteration.
- In `Subtractor`, iterate from 1 to 100 and decrement the count variable by 1 on each iteration.

- Print the final value of the count variable.
- What would the ideal value of the count variable be?
- What is the actual value of the count variable?
- Try to add some delay in the `Adder` and `Subtractor` classes using inspiration from the code below. What is the value of the count variable now?

Simple implementation

First, here is the `Count` class:

```
class Count:
    def __init__(self):
        self.count = 0

    def increment(self):
        self.count += 1

    def decrement(self):
        self.count -= 1

    def get_count(self):
        return self.count
```

We'll start with multi-threading without synchronisation. We'll see how the threads interfere with each other and how we can solve this problem using synchronisation.

```
from threading import Thread

class Adder(Thread):
    def __init__(self, count):
        self.count = count

    def run(self):
        for i in range(100):
            self.count.increment()
```

In the above code, we have a class `Adder` that extends the `Thread` class and overrides the `run` method. The `run` method is called when the thread is started. In the `run` method, we

iterate from 1 to 100 and increment the count variable by 1 on each iteration.

Similarly for the `Subtractor` class:

```
class Subtractor(Thread):
    def __init__(self, count):
        self.count = count

    def run(self):
        for i in range(100):
            self.count.decrement()
```

Let's add a runner that creates the `Count` object, the `Adder` object and the `Subtractor` object. It then starts the `Adder` and `Subtractor` threads and waits for them to finish.

```
def main():
    count = Count()
    adder = Adder(count)
    subtractor = Subtractor(count)

    adder.start()
    subtractor.start()

    adder.join()
    subtractor.join()

    print(count.get_count())

if __name__ == '__main__':
    main()
```

Motivation for synchronisation

Running the above code should ideally result in the count variable being 0. However, if you increase the number of iterations in the `Adder` and `Subtractor` classes, you will see that the count variable is not 0. This is because the `Adder` and `Subtractor` classes are running in parallel and are not synchronised. This means that the `Adder` and `Subtractor` classes are not waiting for each other to finish before they start their own iterations. This results in the

count variable being incremented and decremented at the same time, resulting in the count variable not being 0.

The major issue is that multiple threads are accessing a shared resource at the same time.

Characteristics of synchronisation problems

- `Critical section` - A section of code that is accessed by multiple threads. When multiple threads access the same critical section, the result is a synchronisation problem that might yield wrong or inconsistent results.

```
    def run(self):
        for i in range(100):
            self.count.increment() # Critical section
```

- `Race Conditions` - When more than one thread tries to enter the critical section at the same time.
- `Preemption` - When a thread is interrupted by another thread. It could be possible that the interrupted thread is in the middle of a critical section. This could result in the interrupted thread not being able to finish the critical section and yield inconsistent results.

Properties of a good solution

- `Mutual Exclusion` - Only one thread can access the critical section at a time.
- `Progress` - If a thread wants to enter the critical section, it will eventually be able to do so.
- `Bounded Waiting` - If a thread wants to enter the critical section, it will eventually be able to do so, but only after a finite number of other threads have entered the critical section.
- `No busy Waiting` - If a thread wants to enter the critical section, it will not be able to do so until the critical section is free. It has to keep checking if the critical section is free. This is called busy waiting.
- `Notification` - If a thread is waiting to enter the critical section, it should be notified when the critical section is free.

Solutions to synchronisation problems

Mutex Locks

Mutex locks are a way to solve the synchronisation problem. Mutex locks are a way to ensure that only one thread can access a critical section at a time. Mutex locks are also known as `mutual exclusion locks`.

A thread can only access the critical section if it has the lock. If a thread does not have the lock, it cannot access the critical section. If a thread has the lock, it can access the critical section. If a thread has the lock, it can release the lock and allow another thread to access the critical section.

Think of a room with a lock. Only one person can enter the room at a time. If a person has the key, they can enter the room. If a person does not have the key, they cannot enter the room. If a person has the key, they can leave the room and give the key to another person. This is the same as a mutex lock.

Properties of a mutex lock

- `Lock` - A thread can only access the critical section if it has the lock.
- Only one thread can have the lock at a time.
- Other threads cannot access the critical section if a thread has the lock and thus have to wait.
- Lock will automatically be released when the thread exits the critical section.

Using locks in Python

In Python, lock objects, represented by the threading.Lock class, are synchronization mechanisms used to control access to shared resources in a multithreaded environment. Locks help prevent race conditions by allowing only one thread at a time to execute a critical section of code, ensuring data integrity and thread safety.

You start by creating a lock object:

```
from threading import Lock

lock = Lock()
```

You can then use the lock object to lock and unlock the critical section.
A thread acquires a lock using the `acquire()` method and releases it using the `release()` method. The `acquire` method blocks if the lock is currently held by another thread.

```
# Acquiring the lock
lock.acquire()

print("Inside the critical section")

# Releasing the lock
lock.release()
```

Now, you can rewrite the `Adder` and `Subtractor` classes to use locks:

```
class Adder(Thread):
    def __init__(self, count, lock):
        self.count = count
        self.lock = lock

    def run(self):
        for i in range(100):
            self.lock.acquire()
            self.count.increment()
            self.lock.release()

class Subtractor(Thread):
    def __init__(self, count, lock):
        self.count = count
        self.lock = lock

    def run(self):
        for i in range(100):
            self.lock.acquire()
            self.count.decrement()
            self.lock.release()
```

Now, the `Adder` and `Subtractor` classes acquire the lock before accessing the critical section and release the lock after accessing the critical section. This ensures that only one thread can access the critical section at a time.

```
def main():
    count = Count()
    lock = Lock()
    adder = Adder(count, lock)
    subtractor = Subtractor(count, lock)

    adder.start()
    subtractor.start()

    adder.join()
    subtractor.join()

    print(count.get_count())
```

Reentrant locks

A reentrant lock is a lock that can be acquired multiple times by the same thread.
This can be useful in scenarios where a function calls another function that also needs the lock. If the lock was not reentrant, the thread would block itself when it tries to acquire the lock for the second time.

```
from threading import RLock

def function(lock):
    lock.acquire()
    print("Inside function")
    lock.release()

def main():
    lock = RLock()

    lock.acquire()
    print("Inside main")

    function(lock)

    lock.release()

if __name__ == '__main__':
    main()
```

Context Managers

Context managers in Python provide a convenient way to manage resources, such as acquiring and releasing locks, opening and closing files, or connecting and disconnecting from a network. They are typically used with the with statement and offer a clean and readable way to handle resource management, ensuring that resources are properly acquired and released.

Context managers are commonly employed for acquiring and releasing locks in a synchronized manner. The `with` statement ensures that a lock is acquired before entering a critical section and released when exiting the section. A context manager must implement two methods: `__enter__()` and `__exit__()`. The `__enter__` method is called at the beginning of the with block, and `__exit__` is called at the end. Under the hood, the `with` statement is equivalent to a `try` and `finally` block. The `__enter__` method corresponds to the code in the `try` block, and the `__exit__` method corresponds to the code in the `finally` block.

Context managers ensure mutual exclusion, allowing only one thread at a time to access a

critical section. This prevents conflicts and data corruption that may arise when multiple threads access shared resources simultaneously. By using context managers, the with statement handles acquiring and releasing locks, minimizing the chances of deadlocks. Locks are automatically released, even if an exception occurs within the critical section.

```
from threading import Lock

lock = Lock()

with lock:
    print("Inside the critical section")
```

You can rewrite the `Adder` and `Subtractor` classes to use context managers:

```
class Adder(Thread):
    def __init__(self, count, lock):
        self.count = count
        self.lock = lock

    def run(self):
        for i in range(100):
            with self.lock:
                self.count.increment()

class Subtractor(Thread):
    def __init__(self, count, lock):
        self.count = count
        self.lock = lock

    def run(self):
        for i in range(100):
            with self.lock:
                self.count.decrement()
```

Atomic and Concurrent Data Types

In concurrent programming, atomic and concurrent data structures play a crucial role in managing shared resources among multiple threads or processes. These structures ensure

that operations on shared data are performed atomically, without interference from other concurrent operations.

An atomic operation is an operation that appears to be executed in a single, uninterruptible step. In the context of concurrent programming, atomic operations are essential to prevent race conditions and ensure data integrity. Atomic operations are indivisible and are not interrupted by other concurrent operations.

A concurrent data structure is a data structure that can be accessed by multiple threads or processes concurrently. Concurrent data structures are designed to be thread-safe and ensure that operations on shared data are performed atomically. They are typically implemented using locks or other synchronisation mechanisms to ensure mutual exclusion.

Python does not have built-in support for atomic data types. There are some thread-safe data structures in Python, such as the `Queue` class. It is a synchronised queue that allows multiple threads to safely access the queue without the need for locks. The `Queue` class is implemented using locks and condition variables to ensure that operations on the queue are performed atomically.