# **THREADS IN PYTHON**

**SINGLE TASKING VS MULTI-TASKING :**

**Single-tasking**:

Single-tasking refers to the execution model where only one task or thread is active at any given time. In this model, the CPU executes tasks sequentially, one after another. If a task is blocked or waiting for some event (e.g., I/O operation), the CPU remains idle until that task completes.

***Real-time Example****: Imagine you are baking a cake. In a single-tasking scenario, you would complete each step of the baking process one after another. For instance, you would mix the ingredients, then put the cake in the oven, and wait for it to bake. Only after the baking is complete would you move on to the next task, such as decorating the cake.*

**Multi-tasking:**

Multi-tasking, on the other hand, involves executing multiple tasks or threads concurrently. In this model, the CPU switches between tasks or threads rapidly, giving the illusion of simultaneous execution. Each task or thread gets a slice of CPU time to execute its operations, and the scheduler manages the switching between them.

**Real-time Example:** Consider a chef working in a busy kitchen. They might be simultaneously preparing multiple dishes. For instance, while one dish is sautéing on the stove, they might be chopping vegetables for another dish, and at the same time, a third dish might be baking in the oven. Each task (preparing a dish) progresses concurrently with others, and the chef switches between tasks based on what needs attention at any given moment.

🡺Single-tasking with threads involves executing only one thread at a time, while other threads wait for their turn.

🡺Multi-tasking with threads involves executing multiple threads concurrently, with the operating system or a scheduler managing their execution.

**ADVANTAGES OF THREADS**

**Doing More Than One Thing at a Time**: Threads allow your computer to work on multiple tasks simultaneously. Just like you can talk on the phone while cooking dinner, threads let your program handle multiple jobs at once.

**Speeding Things Up**: When one task is waiting for something to happen (like reading from a file or waiting for a network response), threads let other tasks keep working. This helps your program get things done faster overall.

**Better User Experience:** Threads make your program more responsive. Imagine if a webpage took forever to load because it couldn't do anything else while waiting for images to download. Threads let your program continue to respond to user input while it's busy with other tasks.

**Organizing Tasks:** Threads help you organize your program into smaller, more manageable parts. Just like having different people handle different works around the house, threads make it easier to divide up the work and keep things running smoothly.

**Making the Most of Your Computer**: Threads help your computer use its resources efficiently. By keeping the CPU busy with useful tasks, threads make sure your computer isn't just sitting around doing nothing when it could be getting things done.

--------------------------------------------------------------------- -----------------------------------------------------------------------------------------------------------------------------------

To create a thread in Python, you typically follow these steps:

1. **Define a function:** Define a function that represents the task you want the thread to execute.

2. **Create a Thread object:** Use the `Thread` class from the `threading` module to create a thread object. Pass the function you defined in step 1 as the `target` argument to the `Thread` constructor.

3. **Start the thread:** Call the `start()` method on the thread object to start the execution of the thread.

Here's a simple example demonstrating how to create a thread in Python:

```python

import threading

import time

# Define a function that will be executed by the thread

def print\_numbers():

for i in range(5):

print(i)

time.sleep(1)

# Create a thread

thread = threading.Thread(target=print\_numbers)

# Start the thread

thread.start()

```

In this example:

- We import the `threading` module, which provides support for working with threads in Python.

- We define a function named `print\_numbers()` that prints numbers from 0 to 4 with a delay of 1 second between each print.

- We create a thread object named `thread` using the `Thread` class from the `threading` module. We specify the `target` argument as `print\_numbers`, which means this function will be executed by the thread.

- We start the execution of the thread by calling the `start()` method on the `thread` object.

This creates a new thread that runs concurrently with the main thread, executing the `print\_numbers()` function.

**here are the steps outlined in the code from importing the threading module:**

**Importing the threading module:**

**import threading**

This line imports the threading module, which provides support for creating and working with threads in Python.

**Defining the function executed by the thread:**

**def print\_numbers():**

**for i in range(5):**

**print(i)**

**time.sleep(1)**

This defines a function named print\_numbers() that prints numbers from 0 to 4 with a 1-second delay between each print statement. This function will be executed by the thread.

**Creating the thread:**

**thread = threading.Thread(target=print\_numbers)**

This line creates a thread object named thread using the Thread class from the threading module. The target parameter specifies the function that the thread will execute, which in this case is print\_numbers.

**Starting the thread:**

**thread.start()**

This line starts the execution of the thread created in the previous step. Once started, the thread will begin executing the print\_numbers() function concurrently with the main thread.

**WITH OBJECT ORIENTED PROGRAMMING :**

from threading import \*

class Hello(Thread):

    def run(self):

        for i in range(500):

            print("Hello")

class Hi(Thread):

    def run(self):

        for i in range(500):

            print("Hi")

t1 = Hello()

t2 = Hi()

t1.start()

t2.start()

t1.join()

t2.join()

print("Bye")

**Thread synchronization**

Thread synchronization in Python refers to the coordination of multiple threads to ensure that they execute in a predictable and safe manner, particularly when accessing shared resources or critical sections of code. Without proper synchronization, concurrent threads may interfere with each other's execution, leading to race conditions, data corruption, or other unexpected behavior.

There are several techniques for thread synchronization in Python, including:

**Locks (Mutexes):** Locks, also known as mutexes (mutual exclusion), are the most common synchronization primitive. They allow only one thread to acquire the lock at a time, preventing other threads from accessing the shared resource until the lock is released. Python's threading.Lock class provides a simple way to implement locks.

**Semaphores:** Semaphores are similar to locks but allow a specified number of threads to access a resource simultaneously. They maintain a counter that decrements each time a thread acquires the semaphore and increments when it releases it. Python's threading.Semaphore class implements semaphores.

**Condition Variables:** Condition variables are synchronization primitives used to coordinate threads based on certain conditions. Threads can wait on a condition variable until a specific condition is met, allowing them to synchronize their execution. Python's threading.Condition class provides condition variables.

**Events:** Events are synchronization primitives that allow threads to wait for a particular event to occur. Once the event is set, waiting threads are notified and can proceed with their execution. Python's threading.Event class implements events.

**Thread-safe Data Structures:** Python provides thread-safe versions of certain data structures in the queue module, such as Queue, LifoQueue, and PriorityQueue, which can be used for inter-thread communication without the need for explicit synchronization.

**Thread Pooling:** Thread pooling techniques, such as using ThreadPoolExecutor from the concurrent.futures module, can simplify thread management and improve performance by reusing a pool of threads for executing tasks.

**DEADLOCKS**

Deadlock in threads occurs when two or more threads are blocked indefinitely, waiting for each other to release resources that they need in order to proceed. This situation can arise when multiple threads compete for the same resources and acquire them in a way that creates a circular dependency, preventing any of the threads from making progress.

**(OR)**

**Definition:** Deadlock occurs in a multithreaded program when two or more threads are blocked indefinitely, each waiting for the other to release a resource that it needs in order to proceed.

**Resource Competition**: Deadlock typically arises when multiple threads compete for the same resources, such as locks or mutexes, and acquire them in a way that creates a circular dependency.

Necessary Conditions:

Deadlock requires four necessary conditions to be present simultaneously:

**Mutual Exclusion:** At least one resource must be held in a non-sharable mode.

**Hold and Wait:** A thread must hold at least one resource and be waiting to acquire additional resources that are held by other threads.

**No Preemption:** Resources cannot be forcibly taken away from threads.

**Circular Wait:** There must exist a circular chain of threads, each waiting for a resource held by the next thread in the chain.

Example: An example of a deadlock scenario in Python involves two or more threads each holding one resource and waiting for the other resource to be released. This creates a situation where none of the threads can make progress.

**Detection and Prevention:** Detecting and resolving deadlocks can be challenging. Strategies for prevention include careful ordering of lock acquisition, using timeouts for lock acquisition, and minimizing the use of shared resources. Tools such as deadlock detection algorithms and thread profilers can help identify and resolve deadlock issues.

**Mitigation:** In Python, the threading module provides primitives such as locks, semaphores, and condition variables for thread synchronization. Proper use of these primitives, along with good design practices, can help mitigate the risk of deadlocks.

**Debugging:** Debugging deadlocks often involves analyzing thread interactions, identifying potential circular wait conditions, and using tools such as thread dumps or stack traces to diagnose the problem.

***Understanding deadlock concepts and being able to identify and prevent them is crucial for writing reliable and robust multithreaded programs in Python. It requires careful consideration of resource dependencies and synchronization mechanisms to ensure proper thread coordination and avoid deadlock situations.***

**Inter thread communication in python**

In Python, you can achieve inter-thread communication using various mechanisms provided by the threading module. Some common approaches include:

**Queues:** The queue.Queue class is thread-safe and can be used to pass messages or data between threads.

import threading

import queue

def producer(queue):

    for i in range(5):

        queue.put(i)

        print("Produced", i)

def consumer(queue):

    while True:

        item = queue.get()

        if item is None:

            break

        print("Consumed", item)

q = queue.Queue()  # Using Queue directly

producer\_thread = threading.Thread(target=producer, args=(q,))

consumer\_thread = threading.Thread(target=consumer, args=(q,))

producer\_thread.start()

consumer\_thread.start()

producer\_thread.join()

q.put(None)  # signal the consumer thread to exit

consumer\_thread.join()

**Condition Variables**: The threading.Condition class provides a way for threads to synchronize based on the internal state.

import threading

import time

def producer(cv):

    with cv:

        print("Producing...")

        time.sleep(2)

        print("Produced")

        cv.notify()

def consumer(cv):

    with cv:

        print("Consumer waiting...")

        cv.wait(5)

        print("Consumed")

cv = threading.Condition()

producer\_thread = threading.Thread(target=producer, args=(cv,))

consumer\_thread = threading.Thread(target=consumer, args=(cv,))

producer\_thread.start()

consumer\_thread.start()

producer\_thread.join()

consumer\_thread.join()

**Events:** The threading.Event class can be used to signal between threads.

import threading

import time

def worker(event):

    print("Worker waiting for event")

    event.wait()

    print("Worker finished waiting")

event = threading.Event()

thread = threading.Thread(target=worker, args=(event,))

thread.start()

print("Main thread sleeping")

time.sleep(2)

print("Main thread setting event")

event.set()

thread.join()

**Daemon Threads:**

In Python's threading module, threads can be either daemon threads or non-daemon threads. Here's a brief note on daemon threads in Python:

Daemon Threads:

1. Daemon threads are threads that run in the background and do not prevent the program from exiting.
2. They are typically used for tasks that need to run indefinitely while the program is still active, such as monitoring or background tasks.
3. When the main program exits, daemon threads are terminated automatically, regardless of their current state.
4. Daemon threads are created by setting the daemon attribute to True before starting the thread.

You can check if a thread is a daemon thread using the isDaemon() method.

------------------------- ----------------------------------------

import threading

def daemon\_task():

    while True:

        print("Daemon thread is running...")

daemon\_thread = threading.Thread(target=daemon\_task)

daemon\_thread.daemon = True  # Set as daemon thread

daemon\_thread.start()

**Main Thread Behavior:**

* When the main thread exits, it waits for all non-daemon threads to complete before terminating.
* Daemon threads, on the other hand, are abruptly terminated when the main program exits, even if they are in the middle of execution.

**Considerations:**

* Daemon threads are useful for tasks that should not prevent the program from exiting, such as cleanup operations or background services.
* Be cautious when using daemon threads, as they may terminate abruptly, potentially causing resource leaks or leaving tasks unfinished.
* Daemon threads should not be used for tasks that involve critical data processing or must be completed reliably.