CHENNAI INSTITUTE OF TECHNOLOGY
(Autonomous)

AD4305 PARALLEL PROGRAMMING THROUGH PYTHON

UNIT II MULTI-THREADING AND MULTI-PROCESSING IN PYTHON

Understanding Python's Global Interpreter Lock (GIL)-Creating and managing threads using the threading Module-Using the multiprocessing module for CPU-bound Tasks-Synchronization primitives, shared memory, and managing state

---

1. Understanding Python's Global Interpreter Lock (GIL)

- What is the GIL and why does it exist?
- Effects of the GIL on multi-threaded programs in Python
- Impact of GIL on CPU-bound and I/O-bound tasks
- Workarounds to GIL limitations (e.g., using multiprocessing or C extensions)

Understanding Python's Global Interpreter Lock (GIL)

What is the GIL and Why Does it Exist?

The **Global Interpreter Lock (GIL)** is a mutex (mutual exclusion lock) used by the **CPython** interpreter (the default implementation of Python) to ensure that only one thread executes Python bytecode at a time, even on multi-core systems.

- **Purpose of the GIL**:
    1. **Thread safety**: Python manages objects using a system of reference counting. To avoid race conditions while updating reference counts in a multi-threaded program, the GIL ensures only one thread operates at a time.
    2. **Simplicity**: The GIL simplifies memory management and implementation of Python interpreters. Without it, CPython would need to implement fine-grained locking mechanisms, increasing complexity.
    3. **Legacy reasons**: The GIL has existed since Python's early versions when Python was primarily used for single-threaded applications.

Effects of the GIL on Multi-threaded Programs in Python

- The GIL is a significant bottleneck for multi-threaded programs in Python, especially for **CPU-bound tasks**.
- **CPU-bound tasks** (e.g., numerical computations or image processing):
    - Even if you create multiple threads, only one thread can execute Python bytecode at a time due to the GIL.

    - As a result, multi-threading does not achieve true parallelism on multi-core processors for CPU-bound tasks.

- **I/O-bound tasks** (e.g., file I/O, network requests):
    - The GIL is released during I/O operations. Hence, while one thread is waiting for I/O, another thread can execute.
    - This allows Python's threading module to work well for I/O-bound tasks, even with the GIL.

Impact of GIL on CPU-bound and I/O-bound Tasks

1. **CPU-bound tasks**:
    - Tasks requiring intensive computation are hindered by the GIL.
    - Even on multi-core systems, only one core is utilized effectively, making multi-threading inefficient for these workloads.
    - Example: Performing a large matrix multiplication using multiple threads may run slower than using a single thread.
2. **I/O-bound tasks**:
    - Multi-threading is effective for I/O-bound tasks because the GIL is released during blocking I/O operations, such as file reads, network communication, or database queries.
    - This allows other threads to proceed while the blocked thread waits for the I/O operation to complete.

Workarounds to GIL Limitations

1. **Using the multiprocessing module**:
    - The multiprocessing module creates separate processes instead of threads. Each process runs in its own Python interpreter and has its own GIL.
    - This allows true parallelism on multi-core systems since processes do not share memory or a single GIL.
    - Suitable for CPU-bound tasks.

```
from multiprocessing import Process
```

Example:

```
def worker_function():

    print("Worker process")

if __name__== "__main__":

    processes = [Process(target=worker_function) for _ in range(4)]
    for p in processes:
        p.start()
    for p in processes:
        p.join()
```

2. **Using libraries with C extensions**:
   - Libraries like NumPy and pandas perform operations in optimized C code that releases the GIL during computations.
   - This allows them to achieve true parallelism for heavy computations.
3. **Using a different Python interpreter**:
   - **PyPy**: An alternative Python interpreter that uses Just-In-Time (JIT) compilation and manages concurrency differently.
   - **Jython or IronPython**: These interpreters do not have a GIL but may have other limitations (e.g., no support for CPython-specific libraries).
4. **Implementing concurrent I/O with asyncio**:
   - For I/O-bound tasks, asyncio allows asynchronous programming, which avoids the GIL by using event loops instead of threads.

Example:

```
import asyncio
async def fetch_data():
    await asyncio.sleep(1)
    print("Data fetched")
async def main():
    tasks = [fetch_data() for _ in range(3)]
    await asyncio.gather(*tasks)
asyncio.run(main())
```

5. External parallelism tools:

- Use external parallelism frameworks such as **Dask**, **Ray**, or **joblib** for distributed or parallel computing.

What is the Python Global Interpreter Lock (GIL)

Python Global Interpreter Lock (GIL) is a type of process lock which is used by python whenever it deals with processes. Generally, Python only uses only one thread to execute the set of written statements. This means that in python only one thread will be executed at a time. The performance of the single-threaded process and the multi-threaded process will be the same in python and this is because of GIL in python. We can not achieve multithreading in python because we have global interpreter lock which restricts the threads and works as a single thread.

What problem did the GIL solve for Python :

Python has something that no other language has that is a reference counter. With the help of the reference counter, we can count the total number of references that are made internally in python to assign a value to a data object. Due to this counter, we can count the references and when this count reaches to zero the variable or data object will be released automatically. For Example

```
# Python program showing

# use of reference counter

import sys



geek_var = "Geek"

print(sys.getrefcount(geek_var))



string_gfg = geek_var

print(sys.getrefcount(string_gfg))
```

Output:

```
4

5
```

This reference counter variable needed to be protected, because sometimes two threads increase or decrease its value simultaneously by doing that it may lead to memory leaked so in order to protect thread we add locks to all data structures that are shared across threads but sometimes by adding locks there exists a multiple locks which lead to another problem that is deadlock. In order to avoid memory leaked and deadlocks problem, we used single lock on the interpreter that is Global Interpreter Lock(GIL).

Why was the GIL chosen as the solution :
Python supports C language in the backend and all the related libraries that python have are mostly written in C and C++. Due to GIL, Python provides a better way to deal with thread-safe memory management. Global Interpreter Lock is easy to implement in python as it only needs to provide a single lock to a thread for processing in python. The GIL is simple to implement and was easily added to

Python. It provides a performance increase to single-threaded programs as only one lock needs to be managed.

Impact on multi-threaded Python programs :
When a user writes Python programs or any computer programs then there’s a difference between those that are CPU-bound in their performance and those that are I/O-bound. CPU push the program to its limits by performing many operations simultaneously whereas I/O program had to spend time waiting for Input/Output. For Example
Code 1: CPU bound program that perform simple countdown

```
# Python program showing

# CPU bound program


import time

from threading import Thread


COUNT = 50000000


def countdown(n):

    while n>0:

        n -= 1


start = time.time()

countdown(COUNT)

end = time.time()


print('Time taken in seconds -', end - start)
```

Output:

```
Time taken in seconds - 2.5236213207244873
```

Code 2: Two threads running parallel

```
# Python program showing

# two threads running parallel


import time

from threading import Thread


COUNT = 50000000


def countdown(n):

    while n>0:

        n -= 1


t1 = Thread(target = countdown, args =(COUNT//2, ))

t2 = Thread(target = countdown, args =(COUNT//2, ))


start = time.time()

t1.start()

t2.start()

t1.join()

t2.join()

end = time.time()


print('Time taken in seconds -', end - start)
```

Output:

```
Time taken in seconds - 2.183610439300537
```

As you can see, In the above code two code where CPU bound process and multi-threaded process have the same performance because in CPU bound program because GIL restricts CPU to only work with a single thread. The impact of CPU bound thread and multi-threading will be the same in python.

Why hasn’t the GIL been removed yet :

GIL is not improved as of now because python 2 having GIL implementation and if we change this in python 3 then it will create a problem for us. So instead of removing GIL, we improve the concept of GIL. It’s one of the reasons to not remove the GIL at yet is python heavily depends on C in the backend and C extension heavily depends on the implementation methods of GIL. Although there are many more methods to solve the problems that GIL solve most of them are difficult to implement and can slow down the system.

How to deal with Python’s GIL :

Most of the time we use the multiprocessing to prevent the program from GIL. In this implementation, python provide a different interpreter to each process to run so in this case the single thread is provided to each process in multi-processing.

```
# Python program showing

# multiprocessing



import multiprocessing

import time



COUNT = 50000000



def countdown(n):

    while n>0:

        n -= 1



if _name___== "_main_":

    # creating processes

    start = time.time()

    p1 = multiprocessing.Process(target = countdown, args =(COUNT//2, ))

    p2 = multiprocessing.Process(target = countdown, args =(COUNT//2, ))



    # starting process 1
```

```
p1.start()
# starting process 2
p2.start()


# wait until process 1 is finished
p1.join()
# wait until process 2 is finished
p2.join()
end = time.time()
print('Time taken in seconds -', end - start)
```

Output:

```
Time taken in seconds - 2.5148496627807617
```

As you can see that there is no difference between the time taken by the multi-threaded system and the multi-processing system. This is because a multi-processing system has their own problems to solve. So this will not solve the problem but yes it provides the solution that GIL allows to be performed by python.

Python Global Interpreter Lock (GIL)

Summary: The Global Interpreter Lock (GIL) in Python is a mutex that restricts execution to one thread at a time, impacting performance in multi-threaded applications. This blog explores the GIL's workings, implications, workarounds like multi-processing and asynchronous programming, recent developments, and best practices for optimising Python applications.

Introduction

Python, a popular and versatile programming language, has gained widespread adoption across various domains, from web development to data analysis and artificial intelligence. However, Python's design includes a unique feature known as the Global Interpreter Lock (GIL), which has significant implications for concurrent programming and performance.

Understanding the GIL is crucial for Python developers, especially those dealing with multi-threaded applications. While Python's simplicity and ease of use make it an attractive choice for many, the GIL can present challenges that require careful consideration and strategic planning.

This comprehensive guide aims to clarify the GIL's role in Python, its impact on performance, and how to navigate its limitations.

What is the Global Interpreter Lock (GIL)?

The Global Interpreter Lock (GIL) is a mechanism used in CPython, the most widely used implementation of Python, to ensure thread safety. It is a mutex (mutual exclusion) lock that allows only one thread to execute Python bytecode at a time, even on multi-core processors.

Purpose of the GIL

CPython introduced the GIL to simplify the interpreter’s internals and to enable reference counting for memory management. Reference counting manages memory by tracking the number of references to each object.

When the reference count of an object drops to zero, the system can safely deallocate it. The GIL helps prevent race conditions and ensures thread safety by protecting access to Python objects, allowing only one thread to modify an object at a time.

GIL in Other Implementations

It’s important to note that not all Python implementations use the GIL. For example, Jython (Python on the Java platform) and IronPython (Python for .NET) do not have a GIL and can achieve true multi-threading. However, CPython remains the most popular implementation, and thus, the GIL is a critical consideration for most Python developers.

How the GIL Works

The GIL works by allowing only one thread to hold the lock at a time, ensuring that only one thread can execute Python bytecode at any given moment. When a thread acquires the GIL, it can execute Python code until it releases the lock voluntarily or encounters a blocking operation, such as I/O.

Acquiring and Releasing the GIL

When a thread wants to execute Python code, it must first acquire the GIL. Once a thread acquires the GIL, it can execute its bytecode. The system periodically releases and reacquires the GIL during Python code execution.

The frequency of these releases is controlled by a sys.setcheckinterval() function, which determines the number of bytecode instructions executed before the GIL is released. This mechanism allows other threads to have a chance to acquire the GIL and execute their code.

GIL and I/O Operations

In the case of blocking I/O operations, such as reading from a file or making a network request, the GIL is released, allowing other threads to run.

This means that while one thread is waiting for an I/O operation to complete, other threads can execute Python code. However, if all threads are blocked (e.g., waiting for I/O), the entire process will be blocked until one of the threads becomes unblocked.

Implications of the GIL

The implications of the Global Interpreter Lock (GIL) in Python are profound, affecting how developers approach concurrency and performance. This section will explore the limitations imposed by the GIL, its impact on multi-threaded applications, and the trade-offs involved in using Python for parallel processing.

Limits on Parallelism

The GIL prevents true parallelism in Python, even on multi-core systems. Threads cannot execute Python bytecode simultaneously, limiting the potential performance benefits of multi-threading. This means that Python programs may not fully utilise the capabilities of modern multi-core processors, which can be a significant drawback for CPU-bound applications.

Potential Performance Issues

The GIL can lead to performance issues in CPU-bound tasks, where the majority of the computation happens within the interpreter. In such cases, the GIL can become a bottleneck, and the performance of multi-threaded code may not improve or even degrade compared to single-threaded execution.

For example, if you have multiple threads performing heavy computations, they will compete for the GIL, leading to increased context switching and reduced overall performance.

Blocking Operations

When a thread encounters a blocking operation, such as I/O, the GIL is released, allowing other threads to acquire it and execute. However, if all threads are blocked, the entire process will be blocked until one of the threads becomes unblocked. This can lead to inefficiencies in applications that rely heavily on I/O operations.

Compatibility with C Extensions

Python extensions written in C must be GIL-aware to avoid defeating the purpose of threads. Extensions that are not GIL-aware can still be used, but they may limit the effectiveness of multi-threading.

For instance, if a C extension holds the GIL for an extended period while performing a computation, it can block other Python threads from executing, negating the benefits of multi-threading.

Workarounds and Alternatives

To mitigate the limitations of the Global Interpreter Lock (GIL) in Python, developers have devised various workarounds and alternatives. This section will delve into techniques such as multi-processing, asynchronous programming, and the use of alternative Python implementations to achieve better concurrency and performance

Multi-processing

Instead of using threads, Python’s multiprocessing module allows you to create separate processes, each with its own interpreter and memory space. This approach avoids the GIL and enables true parallelism.

Each process can run on a separate core, allowing for better performance in CPU-bound tasks. However, it comes with the overhead of process creation and inter-process communication, which can be more complex than thread-based communication.

Asynchronous Programming

Python’s asyncio module provides a way to write concurrent code using the async/await syntax. Asynchronous programming allows for efficient I/O-bound concurrency without the need for threads or the GIL.

In this model, you can write code that appears synchronous but is executed asynchronously, allowing for non-blocking operations and improved responsiveness.

Alternative Python Implementations

Other Python implementations, such as Jython (Java), IronPython (.NET), and PyPy, may have different approaches to concurrency and may not be affected by the GIL. For example, Jython allows for true multi-threading since it runs on the Java Virtual Machine (JVM), which does not have a GIL.

Cython

Cython, a superset of Python that allows for the use of static types, provides a way to release the GIL temporarily using the with nogil statement. This can be useful for CPU-bound tasks that can be parallelized using OpenMP directives. By releasing the GIL, you can allow other threads to run while performing computations in Cython.

Recent Developments and Future Directions

The Python community has been actively discussing ways to improve concurrency and address the limitations of the GIL. Some recent developments and future directions include:

PEP 703

PEP 703 is a Python Enhancement Proposal that aims to make the GIL optional in CPython by introducing a build configuration flag (–disable-gil). This would allow running Python code without the GIL and with the necessary changes to make the interpreter thread-safe.

While this proposal is still under discussion, it represents a significant step towards addressing the limitations imposed by the GIL.

Scalable Performance

The Python Software Foundation has funded a project to improve the scalability of Python's performance on multi-core systems. This project aims to reduce the overhead of the GIL and improve the performance of multi-threaded code.

As part of this initiative, researchers are exploring new techniques for managing concurrency and improving the overall efficiency of the Python interpreter.

Concurrent Garbage Collection

Research is ongoing to develop a concurrent garbage collector for CPython that can work alongside the GIL, potentially improving the performance of long-running, CPU-bound tasks. A concurrent garbage collector would allow for more efficient memory management without blocking the execution of threads, leading to better overall performance.

Practical Tips and Best Practices

In this section, we will explore practical tips and best practices for effectively managing the Global Interpreter Lock (GIL) in Python, enabling developers to optimise performance and improve concurrency in their applications.

Profile Your Code

Determine whether your code is I/O-bound or CPU-bound. If it’s I/O-bound, multi-threading can still provide benefits, even with the GIL. If it’s CPU-bound, consider using multi-processing or alternative approaches. Profiling tools like cProfile can help you identify bottlenecks in your code.

Use the Multiprocessing Module

For CPU-bound tasks, the multiprocessing module can provide better performance than multi-threading by avoiding the GIL. This module allows you to create separate processes that can run concurrently on multiple cores, improving performance for compute-intensive tasks.

Leverage Asynchronous Programming

For I/O-bound tasks, use Python’s asyncio module to write efficient concurrent code without the need for threads or the GIL. Asynchronous programming can significantly improve the responsiveness of applications that rely on I/O operations.

Be Aware of GIL-aware Extensions

When using Python extensions written in C, ensure that they are GIL-aware to avoid potential issues. Look for extensions that explicitly state their compatibility with multi-threading to ensure optimal performance.

Stay Informed About Developments

Keep track of the latest developments in the Python community regarding concurrency and the GIL. As new tools and techniques emerge, they may provide better solutions for your specific use case. Engaging with the community through forums and conferences can help you stay updated.

Conclusion

The Global Interpreter Lock (GIL) is a fundamental aspect of CPython’s design that has significant implications for concurrent programming. While it simplifies the interpreter’s internals and ensures thread safety, it also limits the potential for true parallelism and can lead to performance issues in certain scenarios.

To work effectively with the GIL, it’s essential to understand its implications, leverage appropriate workarounds and alternatives, and stay informed about the latest developments in the Python community. By following best practices and staying adaptable, developers can navigate the challenges posed by the GIL and create efficient, concurrent applications in Python.

---

2. Creating and Managing Threads Using the threading Module

- Introduction to the threading module
- Creating threads: using Thread class and target functions
- Thread lifecycle and management
    - Starting and stopping threads
    - Joining threads (join() method)

- Thread communication and sharing data
- Applications of threading in I/O-bound tasks

Creating and Managing Threads Using the threading Module

Introduction to the threading Module

The threading module in Python provides a high-level interface for creating and managing threads. It allows you to run multiple threads concurrently within the same process. While threads in Python do not achieve true parallelism due to the Global Interpreter Lock (GIL), they are highly effective for **I/O-bound tasks**.

Key features of the threading module include:

- Easy creation of threads.
- Synchronization primitives like Lock, RLock, Condition, Event, and Semaphore.
- Thread-safe management of shared resources.

Creating Threads: Using Thread Class and Target Functions

Threads are created using the Thread class. You can define the target function the thread will execute.

Example 1: Creating a Thread with a Target Function

```
import threading


def print_numbers():
    for i in range(5):
        print(f"Number: {i}")


# Create a thread

thread = threading.Thread(target=print_numbers)


# Start the thread

thread.start()


# Wait for the thread to finish

thread.join()
```

Example 2: Subclassing the Thread Class
You can also define a custom thread class by subclassing Thread and overriding the run() method.

```
class CustomThread(threading.Thread):

    def run(self):

        for i in range(5):

            print(f"Custom Thread: {i}")


# Create and start the thread

thread = CustomThread()

thread.start()

thread.join()
```

Thread Lifecycle and Management

Threads in Python follow a specific lifecycle:

1. **New**: A thread object is created but not yet started.
2. **Runnable**: The thread is ready to run but may not be executing.
3. **Running**: The thread is executing its target function.
4. **Blocked/Waiting**: The thread is waiting for a resource or event.
5. **Terminated**: The thread has completed execution.

Starting and Stopping Threads

- **Starting a Thread**: Use the start() method to begin execution of the thread's run() method.
- **Stopping a Thread**: Python does not provide a direct way to stop threads safely. You should design your threads to check for a condition (e.g., a flag) that allows them to terminate gracefully.

Example: Graceful Thread Termination

```
import threading

import time



def worker(stop_event):

    while not stop_event.is_set():

        print("Thread running...")
        time.sleep(1)


# Create an Event object for stopping
stop_event = threading.Event()
thread = threading.Thread(target=worker, args=(stop_event,))


thread.start()
time.sleep(5)  # Let the thread run for 5 seconds
stop_event.set()  # Signal the thread to stop
thread.join()
import threading
import time


def worker(stop_event):
    while not stop_event.is_set():
        print("Thread running...")
        time.sleep(1)


# Create an Event object for stopping
stop_event = threading.Event()
thread = threading.Thread(target=worker, args=(stop_event,))


thread.start()
time.sleep(5)  # Let the thread run for 5 seconds
stop_event.set()  # Signal the thread to stop
thread.join()
```

Joining Threads (join() Method)

The join() method ensures that the main program waits for a thread to complete before proceeding.

Example:

```
import threading


def worker():
    print("Worker thread started")
    for _ in range(3):
        print("Working...")
    print("Worker thread finished")


thread = threading.Thread(target=worker)
thread.start()
thread.join()  # Wait for the thread to complete
print("Main thread continues")
```

Thread Communication and Sharing Data

Threads often need to share data or communicate. This requires careful management to avoid **race conditions**.

1. **Using Lock for Synchronization**
   A Lock ensures that only one thread can access a shared resource at a time.

```
import threading


counter = 0
lock = threading.Lock()


def increment():
    global counter
    for _ in range(100000):
        with lock:  # Acquire and release the lock
            counter += 1
```

```
threads = [threading.Thread(target=increment) for _ in range(2)]
for t in threads:
    t.start()
for t in threads:
    t.join()

print("Final Counter:", counter)
```

Using Queue for Thread Communication
The queue.Queue class is thread-safe and provides an efficient way for threads to communicate.

```
import threading
from queue import Queue

def producer(queue):
    for i in range(5):
        queue.put(i)
        print(f"Produced: {i}")

def consumer(queue):
    while not queue.empty():
        item = queue.get()
        print(f"Consumed: {item}")

q = Queue()
producer_thread = threading.Thread(target=producer, args=(q,))
consumer_thread = threading.Thread(target=consumer, args=(q,))

producer_thread.start()
producer_thread.join()
```

```
consumer_thread.start()

consumer_thread.join()
```

Applications of Threading in I/O-bound Tasks

The threading module is particularly effective for I/O-bound tasks where threads can perform other operations while waiting for I/O to complete.

1. **File I/O**: Threads can read and write files concurrently.
2. **Network I/O**: Threads are used to handle multiple network connections simultaneously (e.g., in a web server or chat application).
3. **Database Operations**: Threads can perform queries concurrently while the database processes them.

Example: Multi-threaded I/O-bound Task

```
import threading

import time

def fetch_data(task_id):

    print(f"Task {task_id} started")

    time.sleep(2)  # Simulating an I/O operation

    print(f"Task {task_id} finished")

threads = [threading.Thread(target=fetch_data, args=(i,)) for i in range(5)]

for t in threads:

    t.start()

for t in threads:

    t.join()
```

The threading module is a powerful tool for concurrent programming in Python, but its use is best suited to I/O-bound tasks where the GIL does not limit performance. For CPU-bound tasks, consider using the multiprocessing module instead.

How to Use threading Module to Create Threads in Python

You may have heard the terms "**parallelization**" or "**concurrency**", which refer to scheduling tasks to run parallelly or concurrently (at the same time) to save time and resources. This is a common practice in asynchronous programming, where coroutines are used to execute tasks concurrently.

Threading in Python is used to run multiple tasks at the same time, hence saving time and resources and increasing efficiency.

Although multi-threading can save time and resources by executing multiple tasks at the same time, using it in code can lead to safety and reliability issues.

In this article, you’ll learn what is threading in Python and how you can use it to make multiple tasks run concurrently.

What is Threading?

Threading, as previously stated, refers to the concurrent execution of multiple tasks in a single process. This is accomplished by utilizing Python’s threading module.

Threads are smaller units of the program that run concurrently and share the same memory space.

How to Create Threads Using the threading Module

Python provides a module called threading that provides a high-level threading interface to create and manage threads in Python programs.

Create and Start a Thread

A thread can be created using the Thread class provided by the threading module. Using this class, you can create an instance of the Thread and then start it using the .start() method.

```
import threading

 # Creating Target Function

def num_gen(num):

    for n in range(num):

        print("Thread: ", n)


# Main Code of the Program

if __name__== "__main__":

    print("Statement: Creating and Starting a Thread.")

    thread = threading.Thread(target=num_gen, args=(3,))

    thread.start()

    print("Statement: Thread Execution Finished.")
```

A thread is created by instantiating the Thread class with a target parameter that takes a callable object in this case, the num_gen function, and an args parameter that accepts a list or tuple of arguments, in this case, 3.

This means that you are telling Thread to run the num_gen() function and pass 3 as an argument.

If you run the code, you’ll get the following output:

```
Statement: Creating and Starting a Thread.
Statement: Thread Execution Finished.
Thread:  0
Thread:  1
Thread:  2
```

You can notice that the **Statement** section of the code has finished before the Thread did. **Why does this happen?**

The thread starts executing concurrently with the main program and the main program does not wait for the thread to finish before continuing its execution. That's why the above code resulted in executing the print statement before the thread was finished.

To understand this, you need to understand the execution flow of the program:

- First, the "Statement: Creating and Starting a Thread." print statement is executed.
- Then the thread is created and started using thread.start().
- The thread starts executing concurrently with the main program.
- The "Statement: Thread Execution Finished." print statement is executed by the main program.
- The thread continues and prints the output.

The thread and the main program run independently that's why their execution order is not fixed.

join() Method – The Saviour

Seeing the above situation, you might have thought then how to suspend the execution of the main program until the thread is finished executing.

Well, the join() method is used in that situation, it doesn't let e**xecute the code further until the current thread terminates**.

```
import threading

# Creating Target Function
def num_gen(num):
    for n in range(num):
        print("Thread: ", n)

# Main Code of the Program
```

```
if __name___== "__main__":
    print("Statement: Creating and Starting a Thread.")
    thread = threading.Thread(target=num_gen, args=(3,))
    thread.start()
    thread.join()
    print("Statement: Thread Execution Finished.")
```

After creating and starting a thread, the join() method is called on the Thread instance (thread). Now run the code, and you’ll get the following output.

```
Statement: Creating and Starting a Thread.
Thread:  0
Thread:  1
Thread:  2
Statement: Thread Execution Finished.
```

As can be seen, the "Statement: Thread Execution Finished." print statement is executed after the thread terminates.

Daemon Threads

Daemon threads run in the background and terminate immediately whether they completed the work or not when the main program exits.

You can make a daemon thread by passing the daemon parameter when instantiating the Thread class. You can pass a boolean value to indicate whether the thread is a daemon (True) or not (False).

```
import threading
import time

def daemon_thread():
    while True:
        print("Daemon thread is running.")
        time.sleep(1)
        print("Daemon thread finished executing.")

if __name___== "__main__":
```

```
thread1 = threading.Thread(target=daemon_thread, daemon=True)


thread1.start()
print("Main program exiting.")
```

A thread is created by instantiating the Thread class passing the daemon_thread function inside it and to mark it as a **daemon thread**, the daemon parameter is set to True.

The daemon_thread() function is an infinite loop that prints a statement, sleeps for one second, and then again prints a statement.

Now when you run the above code, you'll get the following output.

```
Daemon thread is running.Main program exiting.
```

You can see that as soon as the main program exits, the daemon thread terminates.

At the time when the daemon_thread() function enters the loop, the concurrently running main program exits, and the daemon_thread() function never reaches the next print statement as can be seen in the output.

threading.Lock – Avoiding Race Conditions

Threads, as you know, run concurrently in a program. If your program has multiple threads, they may share the same resources or the critical section of the code at the same time, this type of condition is called **race conditions**.

This is where the Lock comes into play, it acts like a synchronization barrier that prevents multiple threads from accessing the particular code or resources simultaneously.

The thread calls the acquire() method to acquire the Lock and the release() method to release the Lock.

```
import threading

# Creating Lock instance
lock = threading.Lock()

data = ""

def read_file():
    global data
    with open("sample.txt", "r") as file:
        for info in file:
            data += "\n" + info

def lock_task():
    lock.acquire()
    read_file()
    lock.release()

if __name__== "__main__":
    thread1 = threading.Thread(target=lock_task)
    thread2 = threading.Thread(target=lock_task)

    thread1.start()
    thread2.start()

    thread1.join()
    thread2.join()

    # Printing the data read from the file
    print(f"Data: {data}")
```

First, a Lock is created using the threading.Lock() and store it inside the lock variable.

An empty string is created (data) for storing the information from both threads concurrently.

The read_file() function is created that reads the information from the sample.txt file and adds it to the data.

The lock_task() function is created and when it is called, the following events occur:

- The lock.acquire() method will acquire the Lock immediately when the lock_task() function is called.
- If the Lock is available, the program will execute the read_file() function.
- After the read_file() function finished executing, the lock.release() method will release the Lock to make it available again for other threads.

Within the if __name___== "_main_" block, two threads are created thread1 and thread2 that both runs the lock_task() function.

Both threads run concurrently and attempt to access and execute the read_file() function at the same time but only one thread can access and enter the read_file() at a time due to the Lock.

The main program waits for both threads to execute completely because of thread1.join() and thread2.join().

Then using the print statement, the information present in the file is printed.

```
Data:

Hello there! Welcome to GeekPython.

Hello there! Welcome to GeekPython.
```

As can be seen in the output, one thread at a time reads the file. However, there were two threads that's why the file was read two times, first by thread1 and then by thread2.

Semaphore Objects in Threading

Semaphore allows you to limit the number of threads that you want to access the shared resources simultaneously. Semaphore has two methods:

- acquire(): Thread can acquire the semaphore if it is available. **When a thread acquires a semaphore, the semaphore's count decrement** if it is greater than zero. If the count is zero, the thread waits until the semaphore is available.
- release(): After using the resources, the **thread releases the semaphore that results in an increment in the count**. This means that shared resources are available.

Semaphore is used to limit access to shared resources, preventing resource exhaustion and ensuring controlled access to resources with limited capacity.

```
import threading

# Creating a semaphore
sem = threading.Semaphore(2)

def thread_task(num):
    print(f"Thread {num}: Waiting")

    # Acquire the semaphore
    sem.acquire()
    print(f"Thread {num}: Acquired the semaphore")

    # Simulate some work
    for _ in range(5):
        print(f"Thread {num}: In process")

    # Release the semaphore when done
    sem.release()
    print(f"Thread {num}: Released the semaphore.")

if __name__== "__main__":
    thread1 = threading.Thread(target=thread_task, args=(1,))
    thread2 = threading.Thread(target=thread_task, args=(2,))
    thread3 = threading.Thread(target=thread_task, args=(3,))

    thread1.start()
    thread2.start()
    thread3.start()

    thread1.join()
    thread2.join()
    thread3.join()

    print("All threads have finished.")
```

In the above code, Semaphore is instantiated with the integer value of 2 which means two threads are allowed to run at the same time.

Three threads are created and all of them use the thread_task() function. But only two threads are allowed to run at the same time, so two threads will access and enter the thread_task() function at the same time, and when any of the threads releases the semaphore, the third thread will acquire the semaphore.

```
Thread 1: Waiting
Thread 1: Acquired the semaphore
Thread 1: In process
Thread 1: In process
Thread 1: In process
Thread 1: In process
Thread 1: In processThread 2: Waiting
Thread 2: Acquired the semaphore

Thread 1: Released the semaphore.
Thread 2: In process
Thread 2: In process
Thread 3: WaitingThread 2: In process
Thread 3: Acquired the semaphore
Thread 3: In process

Thread 2: In process
Thread 2: In process
Thread 2: Released the semaphore.
Thread 3: In process
Thread 3: In process
Thread 3: In process
Thread 3: In process
Thread 3: Released the semaphore.
All threads have finished.
```

Using ThreadPoolExecutor to Execute Tasks from a Pool of Worker Threads

The ThreadPoolExecutor is a part of concurrent.features module that is used to execute multiple tasks concurrently. Using ThreadPoolExecutor, you can run multiple tasks or functions concurrently without having to manually create and manage threads.

```
from concurrent.futures import ThreadPoolExecutor

# Creating pool of 4 threads
executor = ThreadPoolExecutor(max_workers=4)

# Function to evaluate square number
def square_num(num):
    print(f"Square of {num}: {num * num}.")

task1 = executor.submit(square_num, 5)
task2 = executor.submit(square_num, 2)
task3 = executor.submit(square_num, 55)
task5 = executor.submit(square_num, 4)

# Wait for tasks to complete and then shutdown
executor.shutdown()
```

The above code creates a ThreadPoolExecutor with a maximum of 4 worker threads which means the thread pool can have a maximum of 4 worker threads executing the tasks concurrently.

Four tasks are submitted to the ThreadPoolExecutor using the submit method with the square_num() function and various arguments. This will execute the function with specified arguments and prints the output.

In the end, the shutdown method is called, so that ThreadPoolExecutor shutdowns after the tasks are completed and resources are freed.

You don’t have to explicitly call the shutdown method if you create ThreadPoolExecutor using the with statement.

```
from concurrent.futures import ThreadPoolExecutor

# Task
def square_num(num):
    print(f"Square of {num}: {num * num}.")
```

```

# Using ThreadPoolExecutor as context manager
with ThreadPoolExecutor(max_workers=4) as executor:
    task1 = executor.submit(square_num, 5)
    task2 = executor.submit(square_num, 2)
    task3 = executor.submit(square_num, 55)
    task5 = executor.submit(square_num, 4)
```

In the above code, the ThreadPoolExecutor is used with the with statement. When the with block is exited, the ThreadPoolExecutor is automatically shut down and its resources are released.

Both codes will produce the same result.

```
Square of 5: 25.

Square of 2: 4.

Square of 55: 3025.

Square of 4: 16.
```

Common Function in Threading

The threading module provides numerous functions and some of them are explained below.

Getting Main and Current Thread

The threading module has a main_thread() and a current_thread() function which is used to get the **main thread** and the **currently running thread** respectively.

```
import threading

def task():
    for _ in range(2):
        # Getting the current thread name
        print(f"Current Thread: {threading.current_thread().name} is running.")

# Getting the main thread name
print(f"Main thread   : {threading.main_thread().name} started.")
thread1 = threading.Thread(target=task)
```

```
thread2 = threading.Thread(target=task)

thread1.start()
thread2.start()

thread1.join()
thread2.join()
print(f"Main thread   : {threading.main_thread().name} finished.")
```

Because the main_thread() and current_thread() functions return a Thread object, threading.main_thread().name is used to **get the name of the main thread** and threading.current_thread().name is used to **get the name of the current thread**.

```
Main thread   : MainThread started.
Current Thread: Thread-1 (task) is running.
Current Thread: Thread-1 (task) is running.
Current Thread: Thread-2 (task) is running.
Current Thread: Thread-2 (task) is running.
Main thread   : MainThread finished.
```

Monitoring Currently Active Threads

The threading.enumerate() function is used to return the list of Thread objects that are currently running. This includes the main thread even if it is terminated and excludes terminated threads and threads that have not started yet.

If you want to get the number of Thread objects that are currently alive, you can utilize the threading.active_count() function.

```
import threading

def task():
    print(f"Current Thread      : {threading.current_thread().name} is running.")

# Getting the main thread name
print(f"Main thread         : {threading.main_thread().name} started.")
```

```python

threads_list = []

for _ in range(5):
    thread = threading.Thread(target=task)
    thread.start()
    threads_list.append(thread)
    # Getting the active thread count
    print(f"\nActive Thread Count: {threading.active_count()}")

for thread in threads_list:
    thread.join()

print(f"Main thread        : {threading.main_thread().name} finished.")
# Getting the active thread count
print(f"Active Thread Count: {threading.active_count()}")
# Getting the list of active threads
for active in threading.enumerate():
    print(f"Active Thread List: {active.name}")
```

Output

```
Main thread        : MainThread started.
Current Thread     : Thread-1 (task) is running.
Active Thread Count: 2

Current Thread     : Thread-2 (task) is running.
Active Thread Count: 2

Current Thread     : Thread-3 (task) is running.
```

```
Active Thread Count: 2

Current Thread      : Thread-4 (task) is running.

Active Thread Count: 2
Current Thread      : Thread-5 (task) is running.

Active Thread Count: 1
Main thread         : MainThread finished.
Active Thread Count: 1
Active Thread List: MainThread
```

Getting Thread Id

```
import threading
import time

def task():
    print(f"Thread {threading.get_ident()} is running.")
    time.sleep(1)
    print(f"Thread {threading.get_ident()} is terminated.")

print(f"Main thread started.")

threads_list = []

for _ in range(5):
    thread = threading.Thread(target=task)
    thread.start()
    threads_list.append(thread)
```

```

for thread in threads_list:
    thread.join()

print(f"Main thread finished.")
```

Every thread running in a process is assigned an identifier and the threading.get_ident() function is used to retrieve the identifier of the currently running thread.

```
Main thread started.
Thread 9824 is running.
Thread 7188 is running.
Thread 4616 is running.
Thread 3264 is running.
Thread 7716 is running.
Thread 7716 is terminated.
Thread 9824 is terminated.
Thread 7188 is terminated.Thread 4616 is terminated.

Thread 3264 is terminated.
Main thread finished.
```

Conclusion

A thread is a smaller unit in the program that is created using the threading module in Python. Threads are tasks or functions that you can use multiple times in your program to execute concurrently to save time and resources.

---

3. Using the multiprocessing Module for CPU-bound Tasks

- Why multiprocessing is better for CPU-bound tasks (overcoming GIL)
- Overview of the multiprocessing module
- Creating and managing processes
    - Using Process class
    - Process lifecycle and management

- Pool of workers (multiprocessing.Pool)
- Comparing multiprocessing with threading for parallel execution

Using the multiprocessing Module for CPU-bound Tasks

Why Multiprocessing is Better for CPU-bound Tasks (Overcoming GIL)

The **Global Interpreter Lock (GIL)** in CPython ensures that only one thread executes Python bytecode at a time. This makes multi-threading ineffective for **CPU-bound tasks** (tasks that require intensive computation, like numerical processing or image analysis), as threads cannot fully utilize multiple CPU cores.

The **multiprocessing module** solves this problem by:

- **Spawning separate processes**: Each process runs its own Python interpreter with its own GIL, allowing true parallelism.
- **Utilizing multiple CPU cores**: By distributing work across multiple processes, CPU-bound tasks can fully leverage multi-core processors.

Example Scenario:

- **Threading**: Adding numbers in a loop with threads may only use one core due to the GIL.
- **Multiprocessing**: The same task using the multiprocessing module runs on multiple processes, utilizing multiple cores for better performance.

Overview of the multiprocessing Module

The multiprocessing module in Python provides a simple way to create and manage processes for parallel execution. It allows developers to:

- Run functions in **parallel** using multiple processes.
- Share data between processes using mechanisms like **queues**, **pipes**, and **shared memory**.
- Synchronize processes using **locks** and other synchronization primitives.

Creating and Managing Processes

Using the Process Class

The Process class is the foundation of the multiprocessing module. Each Process object represents a separate process running independently.

Key Methods:

1. start(): Starts the process and invokes the target function.
2. join(): Waits for the process to complete.

3. terminate(): Terminates the process forcefully.

Example: Creating and Starting Processes

```
from multiprocessing import Process


def worker_function(name):
    print(f"Process {name} is running")


if __name__== "__main__":
    processes = [Process(target=worker_function, args=(i,)) for i in range(4)]


    for process in processes:
        process.start()  # Start each process


    for process in processes:
        process.join()  # Wait for each process to complete
```

Process Lifecycle and Management

1. **New**: The process is created but not yet started.
2. **Running**: The process is executing its target function.
3. **Blocked**: The process is waiting for resources (e.g., I/O or locks).
4. **Terminated**: The process finishes execution or is terminated manually.

Example: Terminating a Process

```
import time
from multiprocessing import Process


def worker():
    while True:
        print("Running...")
        time.sleep(1)
```

```
if __name___== "__main__":
    p = Process(target=worker)
    p.start()
    time.sleep(5)
    p.terminate()  # Forcefully stop the process
    p.join()
    print("Process terminated")
```

Pool of Workers (multiprocessing.Pool)

The Pool class provides a way to manage a pool of worker processes, which are used to execute tasks in parallel. It simplifies process management and is particularly useful for repetitive or batch tasks.

Key Methods:

1. apply(): Executes a function in one of the worker processes (blocking).
2. apply_async(): Executes a function asynchronously in one worker process.
3. map(): Distributes elements of an iterable across multiple processes.

Example: Using Pool for Parallel Execution

```
from multiprocessing import Pool


def square(n):
    return n * n


if __name___== "__main__":
    with Pool(processes=4) as pool:  # Create a pool with 4 processes
        numbers = [1, 2, 3, 4, 5]
        results = pool.map(square, numbers)  # Apply the square function to each number
        print(results)
```

Benefits of Using Pool:

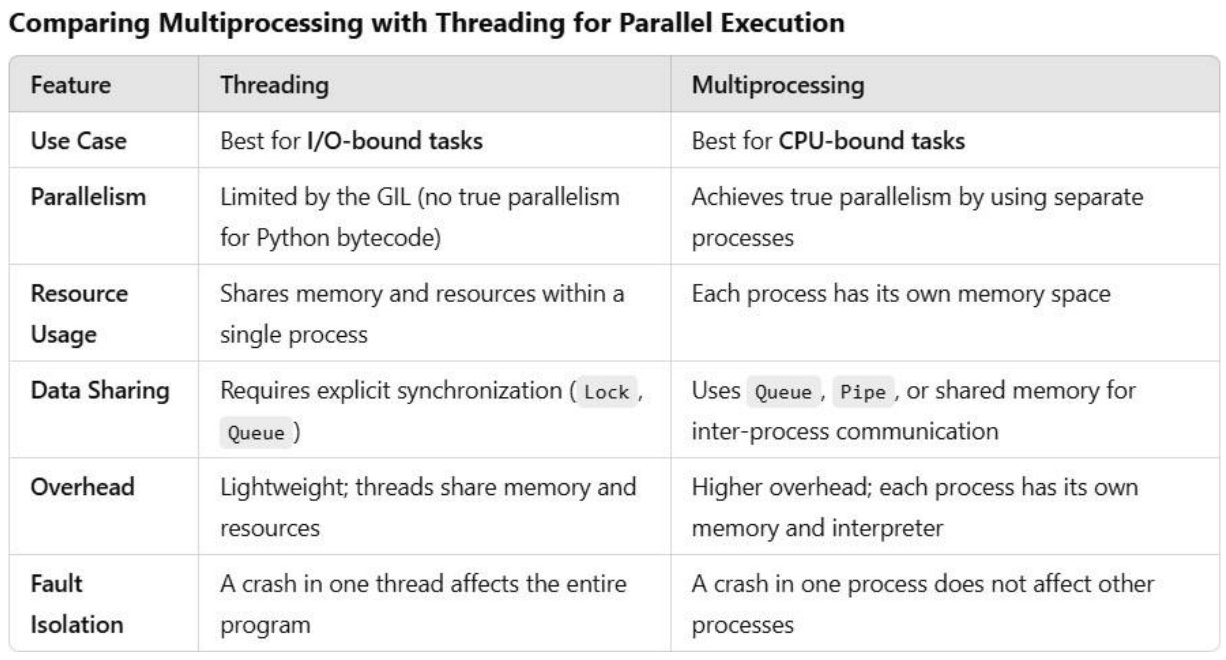
- Automatically manages the lifecycle of worker processes.
- Reduces boilerplate code for process creation and management.

Example: Asynchronous Execution with apply_async

```
from multiprocessing import Pool


def worker_function(x):
    return x * 2


if __name___== "__main__":
    with Pool(processes=3) as pool:
        results = [pool.apply_async(worker_function, args=(i,)) for i in range(5)]
        output = [result.get() for result in results]
        print(output)
```



Comparing Multiprocessing with Threading for Parallel Execution

Feature	Threading	Multiprocessing
Use Case	Best for **I/O-bound tasks**	Best for **CPU-bound tasks**
Parallelism	Limited by the GIL (no true parallelism for Python bytecode)	Achieves true parallelism by using separate processes
Resource Usage	Shares memory and resources within a single process	Each process has its own memory space
Data Sharing	Requires explicit synchronization (`Lock` , `Queue`)	Uses `Queue` , `Pipe` , or shared memory for inter-process communication
Overhead	Lightweight; threads share memory and resources	Higher overhead; each process has its own memory and interpreter
Fault Isolation	A crash in one thread affects the entire program	A crash in one process does not affect other processes

Example: Comparing Threading and Multiprocessing

Threading Example:

```
import threading


def compute_square(n):
    print(f"Square of {n}: {n * n}")
```

```
threads = [threading.Thread(target=compute_square, args=(i,)) for i in range(5)]


for t in threads:
    t.start()


for t in threads:
    t.join()
```

Multiprocessing Example:

```
from multiprocessing import Process


def compute_square(n):
    print(f"Square of {n}: {n * n}")


processes = [Process(target=compute_square, args=(i,)) for i in range(5)]


for p in processes:
    p.start()


for p in processes:
    p.join()
```

Performance Note:

- **Threading**: Inefficient for CPU-intensive tasks due to the GIL.
- **Multiprocessing**: Efficient for CPU-intensive tasks as it uses multiple processes to achieve parallelism.

Summary

- Use **threading** for tasks that involve **I/O-bound operations**, where threads spend time waiting (e.g., reading from disk or network).
- Use **multiprocessing** for tasks that are **CPU-bound**, where heavy computation can benefit from utilizing multiple cores.

- The multiprocessing module is a powerful tool to overcome Python's GIL and achieve true parallelism.

---

4. Synchronization Primitives, Shared Memory, and Managing State

- Synchronization mechanisms in threading and multiprocessing
    - Locks, RLocks, Semaphores, and Events
- Managing shared data:
    - Shared memory using Value and Array in multiprocessing
    - Avoiding race conditions
- Inter-process communication:
    - Pipes and queues (Queue and Pipe classes)
- Best practices for managing state between threads and processes

Synchronization Primitives, Shared Memory, and Managing State

In multi-threaded and multi-process programs, managing shared resources and maintaining a consistent state are critical. Proper synchronization mechanisms are needed to avoid **race conditions** (when multiple threads/processes access and modify shared data concurrently in an unpredictable manner).

Synchronization Mechanisms in Threading and Multiprocessing

The **threading** and **multiprocessing** modules provide synchronization primitives to coordinate access to shared resources.

1. Locks and RLocks

- A **Lock** ensures that only one thread or process can access a shared resource at a time.
- An **RLock** (Reentrant Lock) allows the same thread or process to acquire the lock multiple times without causing a deadlock.

Threading Example: Using Lock

```
import threading


counter = 0

lock = threading.Lock()
```

```
def increment():
    global counter
    for _ in range(100000):
        with lock:  # Lock acquired and released automatically
            counter += 1

threads = [threading.Thread(target=increment) for _ in range(2)]

for t in threads:
    t.start()
for t in threads:
    t.join()

print("Final Counter:", counter)
```

Multiprocessing Example: Using Lock

```
from multiprocessing import Process, Lock

def worker(lock, shared_list):
    with lock:
        for _ in range(5):
            shared_list.append(1)

if _name___== "_main_":
    lock = Lock()
    shared_list = []
    processes = [Process(target=worker, args=(lock, shared_list)) for _ in range(3)]

    for p in processes:
        p.start()
    for p in processes:
        p.join()

    print("Final List Length:", len(shared_list))
```

2. Semaphores

A **Semaphore** allows controlling access to a resource pool with a fixed number of slots. For example, it can limit the number of threads/processes accessing a shared resource simultaneously.

Threading Example: Using Semaphore

```
import threading
import time

semaphore = threading.Semaphore(2)

def worker(name):
    with semaphore:
        print(f"{name} is working...")
        time.sleep(2)
        print(f"{name} finished")

threads = [threading.Thread(target=worker, args=(f"Thread-{i}",)) for i in range(4)]

for t in threads:
    t.start()
for t in threads:
    t.join()
```

3. Events

An **Event** allows one thread or process to signal one or more waiting threads/processes to proceed.

Threading Example: Using Event

```
import threading


event = threading.Event()


def worker():
    print("Waiting for event to be set...")
    event.wait()  # Block until the event is set
    print("Event received! Proceeding with work.")


thread = threading.Thread(target=worker)
thread.start()


print("Main thread setting the event...")
event.set()  # Signal the waiting thread
thread.join()
```

Managing Shared Data

Shared Memory in Multiprocessing

In multiprocessing, each process has its own memory space. To share data between processes, the multiprocessing module provides Value and Array for shared memory.

- **Value**: Represents a single value shared between processes.
- **Array**: Represents an array of shared values.

Example: Using Value and Array

```
from multiprocessing import Process, Value, Array


def increment(shared_value, shared_array):
    with shared_value.get_lock():  # Lock for synchronized access
        shared_value.value += 1
    for i in range(len(shared_array)):
        shared_array[i] += 1
```

```
if __name__== "__main__":
    shared_value = Value('i', 0)  # Shared integer (type 'i')
    shared_array = Array('i', [0, 0, 0])  # Shared array of integers

    processes = [Process(target=increment, args=(shared_value, shared_array)) for _ in range(3)]

    for p in processes:
        p.start()
    for p in processes:
        p.join()

    print("Shared Value:", shared_value.value)
    print("Shared Array:", shared_array[:])
```

Avoiding Race Conditions

Race conditions occur when multiple threads or processes access and modify shared data simultaneously. To avoid race conditions:

1. Use **Locks**, **Semaphores**, or **other synchronization primitives**.
2. Minimize shared state and use local variables where possible.
3. Use thread-safe or process-safe data structures (e.g., queue.Queue or multiprocessing.Queue).

Inter-process Communication

1. Pipes

Pipes provide a simple way for two processes to communicate by sending messages.

Example: Using `Pipe`

```
from multiprocessing import Process, Pipe


def sender(conn):
    conn.send("Hello from sender!")
    conn.close()


def receiver(conn):
    message = conn.recv()
    print(f"Received: {message}")


if _name___== "_main_":
    parent_conn, child_conn = Pipe()
    p1 = Process(target=sender, args=(child_conn,))
    p2 = Process(target=receiver, args=(parent_conn,))


    p1.start()
    p2.start()
    p1.join()
    p2.join()
```

2. Queues

multiprocessing.Queue is a thread- and process-safe queue for sharing data between threads or processes.

Example: Using Queue

```
from multiprocessing import Process, Queue


def producer(queue):
    for i in range(5):
        queue.put(i)
        print(f"Produced: {i}")


def consumer(queue):
    while not queue.empty():
        item = queue.get()
        print(f"Consumed: {item}")


if _name___== "_main_":
    queue = Queue()
    p1 = Process(target=producer, args=(queue,))
    p2 = Process(target=consumer, args=(queue,))

    p1.start()
    p1.join()

    p2.start()
    p2.join()
```

Best Practices for Managing State Between Threads and Processes

1. **Minimize Shared State**:
   - Avoid sharing mutable data unless absolutely necessary.
   - Use local variables instead of shared data.
2. **Use Synchronization Primitives**:
   - Use Lock, RLock, or Semaphore to prevent race conditions.
   - For inter-process communication, use Queue or Pipe.
3. **Prefer Immutable Data Structures**:
   - Immutable data (e.g., tuples) is inherently thread-safe and reduces the risk of synchronization issues.
4. **Avoid Blocking Operations**:
   - Design tasks to avoid unnecessary blocking (e.g., waiting on locks or I/O operations).
5. **Use Thread/Process-safe Libraries**:
   - Libraries like queue.Queue and multiprocessing.Queue handle synchronization internally.
6. **Graceful Termination**:
   - Ensure threads and processes terminate cleanly to avoid resource leaks.

- Use flags (e.g., threading.Event) for controlled shutdowns.

By using these synchronization mechanisms, shared memory tools, and best practices, you can efficiently and safely manage state in multi-threaded and multi-process Python applications.

Synchronization Primitives in Python

Synchronization primitives are mechanisms that help manage access to shared resources in concurrent programming. In Python, the threading and multiprocessing modules provide several synchronization primitives, including **Locks**, **Semaphores**, and **Events**, which prevent race conditions and ensure data consistency.

1. Lock (threading.Lock)

A **Lock** (also known as a **Mutex**) is the simplest synchronization primitive that ensures only one thread at a time can access a shared resource.

How It Works

- A thread acquires the lock before accessing a resource.
- Other threads trying to acquire the lock must wait until the lock is released.

Example

```
import threading

lock = threading.Lock()

counter = 0

def increment():

    global counter

    lock.acquire()  # Acquire the lock

    try:

        counter += 1

    finally:

        lock.release()  # Release the lock

threads = [threading.Thread(target=increment) for _ in range(10)]

for t in threads:

    t.start()

for t in threads:
    t.join()
print("Final Counter Value:", counter)
```

2. Semaphore (threading.Semaphore)

A **Semaphore** is a more flexible synchronization primitive that allows a fixed number of threads to access a resource simultaneously.

How It Works

- A semaphore maintains a counter.
- Threads decrement the counter when acquiring the semaphore.
- If the counter reaches zero, new threads must wait until another thread releases it.

Example

```
import threading
import time
semaphore = threading.Semaphore(3)  # Only 3 threads can access at a time
def worker(thread_id):
    semaphore.acquire()  # Acquire the semaphore
    print(f"Thread {thread_id} is working...")
    time.sleep(2)
    print(f"Thread {thread_id} is done.")
    semaphore.release()  # Release the semaphore
threads = [threading.Thread(target=worker, args=(i,)) for i in range(5)]
for t in threads:
    t.start()
for t in threads:
    t.join()
```

3. Event (threading.Event)

An **Event** is used for signaling between threads. One thread sets an event, and other threads wait for the event to be set before proceeding.

How It Works

- Threads call event.wait() and block until the event is set.

- Another thread calls event.set() to notify waiting threads to continue.

Example

```
import threading

import time

event = threading.Event()

def waiter():

    print("Waiting for event to be set...")

    event.wait()  # Blocks until event is set

    print("Event received! Continuing execution.")

def setter():

    time.sleep(3)

    print("Setting the event.")

    event.set()  # Set the event

thread1 = threading.Thread(target=waiter)

thread2 = threading.Thread(target=setter)

thread1.start()

thread2.start()

thread1.join()

thread2.join()
```

Importance of Synchronization Primitives

1. **Prevent Race Conditions** – Avoids simultaneous modification of shared resources.
2. **Ensure Data Consistency** – Guarantees that multiple threads work on accurate and consistent data.
3. **Manage Resource Access** – Controls how many threads access a resource at a time.
4. **Improve Performance** – Prevents unnecessary contention and deadlocks.

Hybrid System: Using Both Threading and Multiprocessing

This system performs:

- An **I/O-bound task** (simulated by downloading data) using **multithreading**

- A **CPU-bound task** (computing factorial) using **multiprocessing**

This setup allows efficient parallel execution by leveraging **threads** for I/O-bound tasks and **processes** for CPU-bound tasks.

Implementation

```
import threading

import multiprocessing

import time

import requests

# I/O-bound task: Simulates downloading a file

def download_file(url, thread_id):

    print(f"Thread-{thread_id}: Starting download from {url}")

    response = requests.get(url)  # Simulating a network request

    print(f"Thread-{thread_id}: Download complete. Size: {len(response.content)} bytes")

# CPU-bound task: Compute factorial of a large number

def compute_factorial(n):
    print(f"Process: Computing factorial of {n}")
    result = 1
    for i in range(1, n + 1):
        result *= i
    print(f"Process: Factorial computation done for {n}")
# Main function to manage both threading and multiprocessing
def main():
    # Define I/O-bound tasks (URLs to download)
    urls = [
        "https://www.w3.org/WAI/ER/tests/xhtml/testfiles/resources/pdf/dummy.pdf",
        "https://www.learningcontainer.com/wp-content/uploads/2020/05/sample-pdf-file.pdf"
    ]
        # Create and start threads for I/O-bound tasks
    threads = [threading.Thread(target=download_file, args=(url, i)) for i, url in enumerate(urls)]
    for t in threads:
       t.start()
    # Create and start a process for the CPU-bound task
    process = multiprocessing.Process(target=compute_factorial, args=(50000,))
    process.start()
    # Wait for threads and process to finish
    for t in threads:
        t.join()
    process.join()
    print("All tasks completed.")
# Run the system
if _name___== "_main_":
    main()
```

How It Works

1. **I/O-bound Task (Multithreading)**
   - Multiple threads download files concurrently.
   - Uses requests.get() to simulate network delay.
2. **CPU-bound Task (Multiprocessing)**
   - A separate process calculates the factorial of a large number.
   - Uses multiprocessing.Process to avoid GIL limitations.

Expected Output

```
Thread-0: Starting download from
https://www.w3.org/WAI/ER/tests/xhtml/testfiles/resources/pdf/dummy.pdf

Thread-1: Starting download from https://www.learningcontainer.com/wp-
content/uploads/2020/05/sample-pdf-file.pdf

Process: Computing factorial of 50000

Thread-0: Download complete. Size: 13265 bytes

Thread-1: Download complete. Size: 16384 bytes

Process: Factorial computation done for 50000

All tasks completed.
```

Why This Approach?

Task Type	**Technique Used**	**Why?**
I/O-bound	**Threading**	Threads release the GIL during I/O waits, improving efficiency.
CPU-bound	**Multiprocessing**	Processes bypass the GIL, allowing true parallel execution.

Python program that demonstrates **multiprocessing.Value** and **multiprocessing.Array** to share simple data types between multiple processes.

- multiprocessing.Value → Shares a **single value** (e.g., an integer or float) between processes.
- multiprocessing.Array → Shares a **list of values** (e.g., an array of integers or floats) between processes.

Since different processes have separate memory spaces, Python provides **shared memory objects** via **Value** and **Array** to allow processes to communicate.

Implementation

```
import multiprocessing
import time
# Function to increment shared values
def increment(shared_val, shared_arr):
    for _ in range(5):
        with shared_val.get_lock():  # Lock to prevent race conditions
            shared_val.value += 1
           with shared_arr.get_lock():  # Lock to prevent race conditions
            for i in range(len(shared_arr)):
                shared_arr[i] += 1
           time.sleep(1)  # Simulate some processing time
# Main function
def main():
    # Shared integer value (initialized to 0)
    shared_val = multiprocessing.Value('i', 0)  # 'i' = integer type
        # Shared array of integers (initialized to [1, 2, 3])
    shared_arr = multiprocessing.Array('i', [1, 2, 3])  # 'i' = integer type
        # Creating two processes that modify shared data
    p1 = multiprocessing.Process(target=increment, args=(shared_val, shared_arr))
    p2 = multiprocessing.Process(target=increment, args=(shared_val, shared_arr))
    # Start processes
    p1.start()
    p2.start()
    # Wait for both processes to finish
    p1.join()
    p2.join()
    # Print final values
    print(f"Final shared value: {shared_val.value}")
    print(f"Final shared array: {list(shared_arr)}")

# Run the program

if _name___== "_main_":

    main()
```

How It Works

1. **Shared Data Structures:**
   - shared_val (a single integer) starts at **0**.
   - shared_arr (an array of integers) starts as **[1, 2, 3]**.
2. **Two Processes (p1 and p2):**
   - Each process increments shared_val and updates each element in shared_arr.
   - **Locks** (get_lock()) ensure safe updates (avoiding race conditions).
3. **Final Output:**

```
Final shared value: 10

Final shared array: [11, 12, 13]
```

- Each process increments shared_val **5 times** → Total **10**.
- Each element in shared_arr is incremented **10 times** (5 per process).

Inter-Process Communication (IPC) in Python using Pipe and Queue

This program demonstrates **two methods** of communication between processes in Python using the **multiprocessing module**:

- **Pipe** (two-way communication between processes)
- **Queue** (multi-producer, multi-consumer message passing)

Implementation

```
import multiprocessing

import time

# Function to send and receive data using Pipe

def pipe_worker(conn):

    conn.send("Hello from child process!")  # Sending data

    time.sleep(1)  # Simulate some processing
    msg = conn.recv()  # Receiving response
    print(f"Child received: {msg}")
    conn.close()  # Close connection
# Function to send data to the queue
def queue_worker(q):
    for i in range(5):
        q.put(f"Message {i} from child process")
        time.sleep(0.5)  # Simulate delay
# Main function
def main():
    # ------ Using Pipe ------
    parent_conn, child_conn = multiprocessing.Pipe()
    p1 = multiprocessing.Process(target=pipe_worker, args=(child_conn,))
        # Start the process
    p1.start()
    print(f"Parent received: {parent_conn.recv()}")  # Receive from child
    parent_conn.send("Hello from parent process!")  # Send back message
    p1.join()
    # ------ Using Queue ------
    q = multiprocessing.Queue()
    p2 = multiprocessing.Process(target=queue_worker, args=(q,))
        p2.start()
    # Receive messages from the queue
    while p2.is_alive() or not q.empty():
        while not q.empty():
            print(f"Parent received from queue: {q.get()}")
        p2.join()
# Run the program
```

```
if _name___== "_main_":
    main()
```

How It Works

1️⃣Using Pipe (Bidirectional Communication)

- **Parent process** creates a Pipe with two ends: parent_conn and child_conn.
- **Child process** sends a message to the parent and waits for a response.
- **Parent process** receives the message, sends a response, and the child prints it.

Example Output:

```
Parent received: Hello from child process!

Child received: Hello from parent process!
```

2️⃣Using Queue (Multi-Producer, Multi-Consumer)

- **Child process** sends multiple messages to a shared Queue.
- **Parent process** continuously reads from the queue while the child is active.

Example Output:

```
Parent received from queue: Message 0 from child process

Parent received from queue: Message 1 from child process

Parent received from queue: Message 2 from child process

Parent received from queue: Message 3 from child process

Parent received from queue: Message 4 from child process
```

Why Use These Methods?

IPC Method	Best For	Pros	Cons
Pipe	Simple, two-way communication	Fast, low overhead	Only between two processes
Queue	Multi-process communication	Thread-safe, scalable	More overhead than Pipe

---