Multiprocessing and Multithreading

Introduction

- Parallelism vs Concurrency
  - **Concurrency** refers to the ability of different parts of a program, or different programs, to execute *out of order or in partial order* without affecting the final outcome. It's about **managing many things at once**.
  - **Parallelism** refers to the ability to execute multiple tasks or processes *simultaneously*. It's about **doing many things at once**.
- Why parallelism is required?
- Difference between thread and a process?

Sequential vs Concurrent vs Parallel Execution

- Sequential
  - Tasks run one after the other, in strict order. A new task only starts when the previous one finishes.
  - Single thread/process
  - Slowest for multiple tasks
  - do Task A then Task B
- Concurrent
  - Tasks *appear* to run at the same time, often by rapidly switching between them (time-slicing). Progress is made on all tasks.
  - Multithreading (on a single CPU core)
  - Good for I/O-bound tasks
  - start A, while A is waiting (I/O), start B

Sequential vs Concurrent vs Parallel Execution

- Parallel
  - Tasks run truly simultaneously on multiple independent computing resources (like multiple CPU cores).
  - Multiprocessing
  - Best for CPU-bound tasks
  - A and B run simultaneously on different CPU cores.

CPU-bound vs I/O-bound Tasks

- Tasks are often categorized based on where they spend most of their execution time:
- CPU-bound
  - Spends most of its time **calculating** and performing operations on the CPU.
  - CPU resources
  - **Matrix Multiplication**, complex data compression, machine learning model training.
  - **Multiprocessing** (true parallelism)

CPU-bound vs I/O-bound Tasks

- I/O-bound
  - Spends most of its time **waiting** for input/output operations to complete.
  - External operations
  - **File Download**, reading/writing to a database, making a network request.
  - **Multithreading**

- Rule of thumb:
  - Use **threads** for I/O-bound tasks (they can overlap while waiting).
  - Use **processes** (multiprocessing) for CPU-bound tasks

The Global Interpreter Lock (GIL) in Python

- The Global Interpreter Lock (GIL) is a mutex (mutually exclusive lock) that protects access to Python objects, preventing multiple native threads from executing Python bytecodes *simultaneously* in a single process.
- It essentially means that even on a multi-core machine, only one thread can be executing Python bytecode at any given time. This eliminates true parallelism for CPU-bound tasks using standard Python multithreading.
- However, when an I/O operation (like waiting for a file download) is executed, the current thread releases the GIL, allowing another thread to start running Python code. This makes Python multithreading very effective for I/O-bound tasks (concurrency).
- To achieve true parallelism for CPU-bound tasks in Python, you must use the multiprocessing module, which uses separate processes, each with its own Python interpreter and its own GIL.

Multithreading Basics:threading Module

- The threading module is the standard library for implementing multithreading in Python.
- It provides the necessary tools for creating and managing threads.
- Creating and Starting Threads. You can create a thread in two main ways:
  - **Passing a function to the Thread constructor:** Simple and common for short-lived tasks.
  - **Subclassing the Thread class:** Good for more complex threads that might maintain state or have their own methods.

Example 1: Downloading Multiple Files

```python
import threading
import time

def download_file(filename):
    """Simulates an I/O-bound file download."""
    print(f"-> Starting download of {filename}")
    # Simulates the time spent waiting for the network/disk I/O
    time.sleep(2)
    print(f"<- Finished download of {filename}")

files = ["data_1.txt", "video_2.mp4", "image_3.jpg"]
threads = []
start_time = time.time()
```

```python
# 1. Create and start threads
for file in files:
    # target= is the function the thread will execute
    # args= is a tuple of arguments passed to the function
    t = threading.Thread(target=download_file, args=(file,))
    threads.append(t)
    t.start()

# 2. Wait for all threads to complete (Joining)
# The main thread blocks until the joined thread terminates.
for t in threads:
    t.join()

end_time = time.time()
print(f"\nAll downloads complete in {end_time - start_time:.2f} seconds.")

# In a sequential approach, it would take 3 * 2 = 6 seconds.
# With concurrency, it takes just over 2 seconds.
```

Example 2:

```python
# thread_examples.py
import threading
import time

def worker(name, delay=0.1):
    for i in range(3):
        print(f"{name} -> {i}")
        time.sleep(delay)

# Way 1: simple Thread with target
t = threading.Thread(target=worker, args=("T1", 0.2), name="Worker-1")
t.start()
t.join()  # wait

# Way 2: subclassing Thread
class MyThread(threading.Thread):
    def __init__(self, name, delay=0.1):
        super().__init__(name=name)
        self.delay = delay

    def run(self):                # override run()
        worker(self.name, self.delay)

t2 = MyThread("Worker-2", 0.15)
t2.start()
t2.join()
```

Thread Synchronization

- **Race Conditions**
  - A **Race Condition** occurs when two or more threads can access and modify shared data concurrently, and the final outcome depends on the unpredictable order in which the threads execute.
  - The threads are "racing" to complete their operation on the shared resource, and the result depends on which thread finishes its steps first, which can vary from run to run.
- **Locks (threading.Lock) and RLock**
  - To prevent race conditions, we use **synchronization primitives**, most commonly **Locks**.

Thread Synchronization

- **Lock (threading.Lock)**: A basic synchronization primitive. It has two states: **locked** and **unlocked**.
  - A thread calls lock.acquire() to enter a **critical section** (the part of the code that accesses shared resources).
  - If the lock is unlocked, the thread acquires it, and the lock becomes locked.
  - If the lock is already locked, the calling thread is blocked (waits) until the lock is released.
  - The thread calls lock.release() when it's done with the critical section, making the lock available for other threads.

Thread Synchronization

- **Example: Counter Increment Problem (The Classic Race Condition)**
  - The counter increment problem is the quintessential example of a race condition.
- **The Problematic (Unsynchronized) Code**
  - The operation counter = counter + 1 is **not atomic**; it consists of three steps:
  - Read the current value of counter.
  - Calculate the new value (add 1).
  - Write the new value back to counter.

Thread Synchronization

```python
import threading

# Shared, unprotected resource
counter = 0
NUM_INCREMENTS = 100000

def increment_unsynced():
    global counter
    for _ in range(NUM_INCREMENTS):
        # Critical Section (unsynchronized access)
        counter += 1

# Setup and run two threads
t1 = threading.Thread(target=increment_unsynced)
t2 = threading.Thread(target=increment_unsynced)

t1.start()
t2.start()

t1.join()
t2.join()

expected = 2 * NUM_INCREMENTS
print(f"UNSYNCHRONIZED Counter Value: {counter}")
print(f"Expected Value: {expected}")
print(f"Lost increments: {expected - counter}")
# The final value will almost certainly be LESS than the expected 200,000
```

- We use a Lock to ensure that only one thread can execute the critical section (counter += 1) at a time.

```python
import threading

counter = 0
NUM_INCREMENTS = 100000
# Initialize the Lock
lock = threading.Lock()

def increment_synced():
    global counter
    for _ in range(NUM_INCREMENTS):
        # Acquire the lock to enter the critical section
        lock.acquire()
        try:
            # Critical Section: Only one thread can be here at a time
            counter += 1
        finally:
            # Release the lock, even if an error occurred in 'try'
            lock.release()
# Setup and run two threads
t1 = threading.Thread(target=increment_synced)
t2 = threading.Thread(target=increment_synced)

t1.start()
t2.start()

t1.join()
t2.join()

expected = 2 * NUM_INCREMENTS
print(f"\nSYNCHRONIZED Counter Value: {counter}")
print(f"Expected Value: {expected}")
# The final value will reliably be the expected 200,000
```

Thread Creation and Management

- **threading.Thread()**
  - The class used to create a new execution thread. The most common arguments are target (the function to run) and args (arguments for the function).
  - t = threading.Thread(target=worker_func, args=(10, 20), name="Worker")
- **t.start()**
  - Puts the thread into the 'runnable' state. It calls the function specified in target or the run() method of a subclass. **The thread begins execution.**
- **t.join(timeout=None)**
  - **Blocks** the main thread (or the calling thread) until the thread t completes its execution. An optional timeout can be specified.
  -

Thread Creation and Management

- **t.is_alive()**
  - Returns True if the thread is currently executing (has been started and hasn't finished its run() method yet).
  - if t.is_alive(): print("Running...")
- **threading.current_thread()**
  - Returns the **Thread object** corresponding to the caller's thread of control. Useful for getting the thread's name or ID.

Thread Synchronization Primitives

- These are used to control access to shared resources and prevent **race conditions**.
- **threading.Lock()**
  - The most basic synchronization primitive. Only one thread can **acquire** a lock at a time, ensuring mutual exclusion for a **critical section**.
  - lock = threading.Lock() with lock: # Exclusive access starts here shared_resource += 1 # Exclusive access ends here
- **lock.acquire(blocking=True)**
  - Attempts to acquire the lock. If successful, returns True. If blocking is True (default), the thread waits until the lock is available.

Thread Synchronization Primitives

- **lock.release()**
  - Releases the lock. Must only be called by the thread that currently holds the lock.
- **threading.RLock()**
  - A **Reentrant Lock**. Can be acquired multiple times by the **same thread** without causing a deadlock. Must be released the same number of times it was acquired.

```python
import threading
import time

# --- Shared Resources ---
SHARED_COUNTER = 0
NUM_INCREMENTS = 100000
LOCK = threading.Lock()

# --- Function executed by the threads ---
def increment_counter_safe():
    """Increments the global counter safely using a Lock."""
    global SHARED_COUNTER

    for _ in range(NUM_INCREMENTS):
        # 1. Acquire the lock (Critical Section starts)
        with LOCK:
            # This ensures only one thread can execute this line at a time
            SHARED_COUNTER += 1
        # 2. Lock is automatically released here
```

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

    # Resetting the counter for the safe test
    SHARED_COUNTER = 0

    # --- Thread Creation and Management ---

    # Create the threads (target points to the function to run)
    t1 = threading.Thread(target=increment_counter_safe, name="Thread-1")
    t2 = threading.Thread(target=increment_counter_safe, name="Thread-2")

    print("Starting two threads to increment the counter...")
    start_time = time.time()

    # Start the threads
    t1.start()
    t2.start()

    # Join the threads: The main program waits here for both threads to finish
    t1.join()
    t2.join()

    end_time = time.time()

    # --- Results ---

    expected_value = 2 * NUM_INCREMENTS # 2 threads * 100,000 increments

    print("-" * 30)
    print(f"Total execution time: {end_time - start_time:.4f} seconds.")
    print(f"Expected final counter value: {expected_value}")
    print(f"Actual synchronized counter value: {SHARED_COUNTER}")

    # The actual value should equal the expected value, demonstrating successful synchronization
```

Multiprocessing

- The **Process** class is the fundamental building block of Python multiprocessing.
- It allows you to spawn a new process, giving you the ability to run code truly in parallel, bypassing the **Global Interpreter Lock (GIL).**
- You create, start, and join each worker individually.
- Ideal for **heterogeneous tasks** (e.g., Process 1 runs function A, Process 2 runs function B).
- As many as you manually define.
- The process lifecycle involves three main steps:
  - ***defining the task***
  - ***creating the process, and***
  - ***managing its execution.***

Multiprocessing

- **Defining the Task (The Target Function)**
  - A process needs a function to execute. This function is the **target** and must be defined before the process is created.

```
import os
import time

# 1. Define the task function
def worker_function(name, delay):
    """The code that the new process will execute."""
    pid = os.getpid() # Get the Process ID (PID)
    print(f"Process '{name}': Starting with PID {pid}")
    time.sleep(delay)
    print(f"Process '{name}': Finished after {delay} seconds.")
```

Multiprocessing

- **Creating a Process Instance:**
  - You create a Process object, specifying what function to run (target) and what arguments to pass (args).

```
from multiprocessing import Process


# 2. Create the process instance
# target: The function to be executed by the process.
# args: A tuple of arguments for the target function.
p1 = Process(target=worker_function, args=("Task 1", 3))
p2 = Process(target=worker_function, args=("Task 2", 1))

print("Main process is ready to start workers.")
```

Multiprocessing

- **Starting and Waiting (.start() and .join())**
  - The process object is just an abstraction until you explicitly start it.
  - **.start()**: This is the command that actually **spawns the new operating system process**. Control immediately returns to the main program, allowing it to continue execution while the new process runs in the background.
  - **.join()**: When the main program calls .join() on a process, it **blocks** (pauses) its own execution until that specific worker process is complete.

```
# 3. Start the processes
p1.start()
p2.start()

# 4. Wait for the processes to finish
print("Main process is waiting for workers to complete...")
p1.join()
p2.join()

print("Main process finished. All workers are done.")
```

```python
import os
import time
from multiprocessing import Process


def intense_worker(name, duration):
    """Simulates a CPU-bound task by sleeping."""
    pid = os.getpid()
    print(f"[{name}] Starting... PID: {pid}. Will run for {duration}s.")
    start_time = time.time()
    time.sleep(duration)
    end_time = time.time()
    print(f"[{name}] Finished. Total time: {round(end_time - start_time, 2)}s.")
```

```python
if __name__ == "__main__":
    print(f"Main process PID: {os.getpid()}")
    main_start_time = time.time()

    # Define tasks with different durations (5s, 2s, 3s)
    p1 = Process(target=intense_worker, args=("Process 1 (Long)", 5))
    p2 = Process(target=intense_worker, args=("Process 2 (Short)", 2))
    p3 = Process(target=intense_worker, args=("Process 3 (Medium)", 3))

    # Start all processes simultaneously
    p1.start()
    p2.start()
    p3.start()
    print("\n--- All workers launched in parallel ---")

    # Wait for all processes to finish
    p1.join()
    p2.join()
    p3.join()

    # Calculate total time
    total_time = round(time.time() - main_start_time, 2)
    print("\n--- All workers successfully joined ---")
    # Expected output time should be close to 5 seconds, NOT 5+2+3=10 seconds!
    print(f"Total time elapsed for the main program: {total_time} seconds.")
```

The Pool Class: The Batch Runner

- The Pool class provides a high-level, convenient way to manage a fixed number of worker processes.
- It is designed for parallelizing simple, repetitive tasks
  - often described as **map-reduce style operations**
  - where you apply the same function to many different inputs.
- The Pool manages the lifecycle of a set number of workers.
- Ideal for **homogeneous tasks** (e.g., run the same function C 100 times on 100 different inputs).
- **Fixed count,** typically set to os.cpu_count() by default for optimal utilization.
- The **Pool** object is a resource that needs to be properly initialized and, more importantly, cleaned up. If you don't shut it down correctly, it can leave zombie processes running in the background.
- To ensure safe and automatic resource management, we use the **context manager (with statement).**

```python
from multiprocessing import Pool
import os

def check_cpu_count():
    return os.cpu_count()

# Mandatory Syntax: Using the 'with' statement
if __name__ == '__main__':
    num_workers = check_cpu_count()
    print(f"Starting a Pool with {num_workers} worker processes...")

    
    with Pool(processes=num_workers) as pool:
        # Code that uses the pool runs here

        # The pool is automatically terminated and joined when the 'with' block exits.

    print("Pool workers have been safely terminated and joined.")
```

- The **with Pool(...)** as pool:
  - syntax guarantees that pool.close() and pool.join() are called when the block finishes, even if errors occur.
  - This prevents resource leaks and is considered **mandatory** for production code.

- **.map(func, iterable)**
  - This is the simplest and most common method. It takes a single-argument function and an iterable (list, tuple, etc.) and distributes the items from the iterable to the workers.
  - It's a **blocking call**. The main program waits until *all* results are calculated and returned as a single list.
  - The results are guaranteed to be returned in the same order as the input data, regardless of which worker finished first.

```python
# Function that takes one argument
def calculate_square(number):
    """A function suitable for pool.map()"""
    # Simulate a heavy computation
    time.sleep(0.1)
    return number * number

if __name__ == '__main__':
    data = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

    with Pool() as p:
        # p.map splits the 'data' list, sends chunks to workers,
        # waits for results, and returns them in order.
        results = p.map(calculate_square, data)

    print(f"Inputs: {data}")
    print(f"Results: {results}")
    # Output: [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

- **.starmap(func, iterable_of_tuples):**
  - What if your function requires two or more arguments?
  - The iterable must contain **tuples**, where each tuple holds the set of arguments for a single function call.
  - The starmap function "unpacks" (star-unpacks) the tuple into positional arguments for the worker function.

```python
# Function that takes multiple arguments
def calculate_power(base, exponent):
    """A function suitable for pool.starmap()"""
    return base ** exponent

if __name__ == '__main__':
    # Data is a list of (base, exponent) tuples
    data_for_starmap = [(2, 2), (3, 3), (4, 2), (5, 3)]

    with Pool() as p:
        # p.starmap unpacks (2, 2) into base=2, exponent=2, etc.
        results = p.starmap(calculate_power, data_for_starmap)

    print(f"Input Tuples: {data_for_starmap}")
    print(f"Results: {results}")
    # Output: [4, 27, 16, 125]
```

- **.apply_async()**
  - If you don't want the main program to stop and wait for the entire batch to finish (like with .map()), you use .apply_async().
  - It's a **non-blocking call**. It starts the task and immediately returns a **Result object**.
  - The main program can continue running and later call .get() on the result object to retrieve the data when it's ready. This is useful when you have a mix of fast and slow tasks and want to process the fast results immediately.

```
# In pool usage...
if __name__ == '__main__':
    with Pool() as p:
        # Submits the task and gets a non-blocking result object
        result_obj = p.apply_async(calculate_square, args=(10,))

        # Main program does other things here...
        print("Main program is still running...")

        # Retrieve the result, blocking only if the result isn't ready yet
        final_result = result_obj.get()

    print(f"Asynchronous Result: {final_result}")
```

Inter-Process Communication

- Since processes operate in **separate memory spaces**, they cannot simply read or write to each other's variables.
- IPC mechanisms are essential tools that enable processes to exchange data, synchronize, and pass results safely.
- The Queue is the most common and versatile tool for passing messages and results between multiple processes.
- A Queue is a **thread-safe** and **process-safe** FIFO (First-In, First-Out) data structure, similar to a real-world waiting line.
- Data put in by one process can only be retrieved by another process in the order it arrived.
  - **Sending Data:** A process uses the .put(item) method to place an item onto the queue.
  - **Receiving Data:** Another process uses the .get() method to retrieve the next available item.
- The **primary use case is collecting results** from many worker processes back into the main process, or distributing complex tasks from a master process to several workers.

```python
from multiprocessing import Process, Queue

def worker_with_queue(num, result_queue):
    """Worker function that calculates a result and puts it in the queue."""
    result = num * 2
    # The worker sends the result back to the main process
    result_queue.put(result)
    print(f"Worker for {num} finished.")

if __name__ == '__main__':
    # 1. Create the Queue object
    results_q = Queue()
    data = [10, 20, 30]

    processes = []
    for d in data:
        # Pass the Queue object as an argument to the worker
        p = Process(target=worker_with_queue, args=(d, results_q))
        processes.append(p)
        p.start()

    for p in processes:
        p.join()

    final_results = []
    # 2. Retrieve all results from the queue
    while not results_q.empty():
        final_results.append(results_q.get())

    print(f"Final results collected: {final_results}")
    # Output: Final results collected: [20, 40, 60] (Order is not guaranteed due to parallelism)
```

Inter-Process Communication

- A Pipe is a simpler, more direct channel for communication when you only need to connect **two** processes.
- A Pipe creates two connection objects: a **sender** (or parent) and a **receiver** (or child). Data flows through the pipe between these two points.
  - conn1, conn2 = Pipe() creates the two endpoints.
  - One end uses .send(item) and the other uses .recv().
  - It's a two-way connection by default, meaning both ends can send and receive.
- **Use Case:** Ideal for simple, dedicated communication, such as sending a large object from a master process to a single specific worker, or pinging a worker to check its status.

```python
from multiprocessing import Process, Pipe

# The function executed by the child process
def child_worker(conn_to_parent):
    print("Child: Waiting to receive data from parent...")
    # 1. Receive data from the parent
    data_received = conn_to_parent.recv()
    print(f"Child: Received message: '{data_received}'")
    # 2. Process the data
    processed_data = data_received.upper() + " PROCESSED!"
    # 3. Send the result back to the parent
    conn_to_parent.send(processed_data)
    print("Child: Sent processed data back to parent and exiting.")
    # Close the connection when done
    conn_to_parent.close()

if __name__ == '__main__':
    # 1. Create the Pipe: pipe() returns two connection objects
    parent_conn, child_conn = Pipe()
    # parent_conn is the 'parent' end (used by the main process)
    # child_conn is the 'child' end (passed to the worker process)
    # 2. Create and start the child process
    p = Process(target=child_worker, args=(child_conn,))
    p.start()
    initial_message = "hello from the main process"
    print(f"Parent: Sending message: '{initial_message}'")
    # 3. Parent sends data to the child
    parent_conn.send(initial_message)
    # 4. Parent waits to receive data from the child
    print("Parent: Waiting for the processed result...")
    # The .recv() call blocks until data is available
    result_from_child = parent_conn.recv()
    print(f"Parent: Received final result: '{result_from_child}'")
    # 5. Wait for the child process to terminate
    p.join()
    # Close the parent's connection end
    parent_conn.close()
    print("\nPipe communication complete.")
```