Federico Cinelli

CS-510 : Operating Systems Principles

September 13, 2025

**3-2 Activity: Implementing Threads Documentation**

This program demonstrates the management of threads in Python using the threading module. Two separate tasks are defined: one calculates the sum of integers from 1 to 100, and the other generates a random list of numbers and determines its maximum value. Each task is assigned to its own thread, which is created using the threading.Thread() class. The start() method begins with execution of the threads, while the join() method ensures that the main program waits for both tasks to complete before continuing. This approach allows both operations to run concurrently, simulating parallel execution, with short delays introduced by time.sleep(1) to mimic computational work and make the behavior of multithreading more observable.

To execute the program, Python 3 must be installed on the system. The source code should be saved into a file, such as threads.py. The user can then navigate to the file’s directory through a terminal or command prompt and run the program with the command python threads.py. Upon execution, the program outputs the computed sum of numbers from 1 to 100, the randomly generated list of integers, the maximum value found in that list, and finally a confirmation message indicating that both threads have finished their execution.

From an operating system perspective, when the start() method is called, the program requests the creation of a new thread within the process. The operating system allocates resources for the thread and creates a Thread Control Block (TCB), which maintains the thread’s state information, including the program counter, stack pointer, register values, and scheduling details. The scheduler then determines when each thread runs, often switching between threads rapidly to give the illusion of true parallelism. In Python, however, threads are subject to the Global Interpreter Lock (GIL), which restricts simultaneous execution of Python bytecode. As a result, while Python threading does not achieve true parallelism in CPU-bound tasks, it remains useful for tasks involving delays, waiting, or input/output operations.

Several challenges may arise in the implementation of multithreaded programs. The Global Interpreter Lock can limit the performance benefits of threading for computationally intensive workloads. Additionally, because thread scheduling is managed by the operating system, the order of outputs may vary between executions, which can be confusing for beginners. More advanced multithreaded applications also risk encountering race conditions or synchronization issues when threads share resources, requiring the use of locks or other concurrency control mechanisms. In this program, the tasks are independent and do not share data, which eliminates the need for synchronization and simplifies implementation.