Multithreading in Python: The Ultimate Guide (with Coding Examples)

Outline: -

- What's concurrency?
- What's the difference between concurrency and parallelism?
- What's the difference between processes and threads?
- What's multiprocessing in Python?
- What's multithreading in Python?
- What's the Global Interpreter Lock?
- Thread Synchronization

What Is Concurrency?

Before we jump into the multithreading details, let's compare concurrent programming to sequential programming.

In sequential computing, the components of a program are executed step-by-step to produce correct results; however, in concurrent computing, different program components are independent or semi-independent. Therefore, a processor can run components in different states independently and simultaneously. The main advantage of concurrency is in improving a program's runtime, which means that since the processor runs the independent tasks simultaneously, less time is necessary for the processor to run the entire program and accomplish the main task.

What's the Difference Between Concurrency and Parallelism?

Concurrency is a condition wherein two or more tasks can be initiated and completed in overlapping periods on a single processor and core. Parallelism is a condition wherein multiple tasks or distributed parts of a task run independently and simultaneously on multiple processors. So, parallelism isn't possible on machines with a single processor and a single core.

Imagine two queues of customers; concurrency means a single cashier serves customers by switching between two queues. Parallelism means two cashiers simultaneously serve the two queues of customers.

What's the Difference Between Processes and Threads?

A process is a program in execution with its own address space, memory, data stack, etc. The operating system allocates resources to the processes and manages the processes' execution by assigning the CPU time to the different executing processes, according to any scheduling strategy.

Threads are light weight process and they do not require much memory overhead However; they execute within the same process and share the same context. Therefore, sharing information or communicating with the other threads is more accessible than if they were separate processes.

Multiprocessing in Python

Multiprocessing refers to the ability of a system to support more than one processor at the same time. Applications in a multiprocessing system are broken to smaller routines that run independently. Take care about the problems appears in spyder.

- Communication between processes (In multiprocessing, any newly created process will do following: 1-run independently, 2-have their own memory space.)

-See examples about multiprocessing

Multithreading in Python

A thread is a separate flow of execution. This means that your program will have two things happening at once. But for most Python 3 implementations the different threads do not actually execute at the same time: they merely appear to.

It's tempting to think of threading as having two (or more) different processors running on your program, each one doing an independent task at the same time. That's almost right. But for python there are some limitations.

What's the Global Interpreter Lock?

Python virtual machine is not a thread-safe interpreter, meaning that the interpreter can execute only one thread at any given moment. This limitation is enforced by the Python Global Interpreter Lock (GIL), GIL ensures that only one thread runs within the same process at the same time on a single processor.

Basically, threading may not speed up all tasks. The I/O-bound tasks that spend much of their time waiting for external events have a better chance of taking advantage of threading than CPU-bound tasks.

-This make question about how to speed up your program???

NOTE

Python comes with two built-in modules for implementing multithreading programs, including the `thread`, and `threading` modules. The `thread` and `threading` modules provide useful features for creating and managing threads. However, in this tutorial, we'll focus on the `threading` module, which is a much-improved, high-level module for implementing serious multithreading programs.

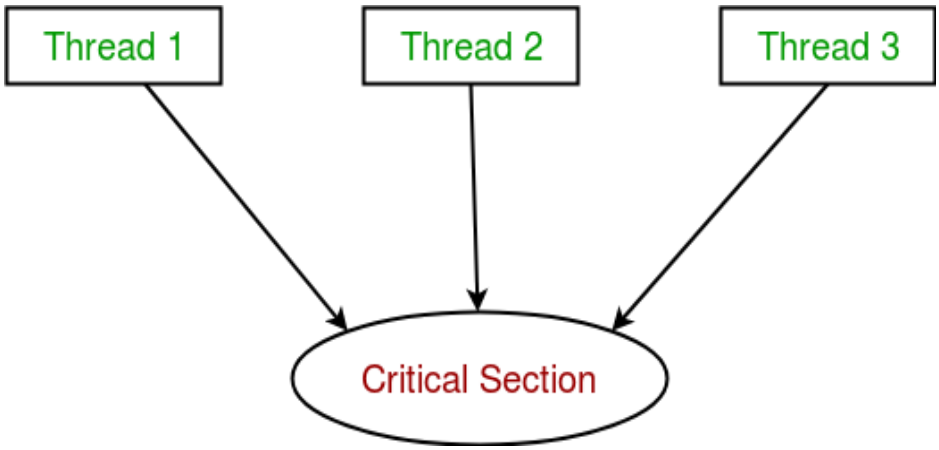
-See examples about multithreading.

Thread Synchronization

Thread synchronization is defined as a mechanism which ensures that two or more concurrent threads do not simultaneously execute some particular program segment known as critical section.

Critical section refers to the parts of the program where the shared resource is accessed.

For example, in the diagram below, 3 threads try to access shared resource or critical section at the same time.



Concurrent accesses to shared resource can lead to race condition.

A race condition occurs when two or more threads can access shared data and they try to change it at the same time. As a result, the values of variables may be unpredictable and vary depending on the timings of context switches of the processes.

-See the code that discuss the problem of race condition(05.py)

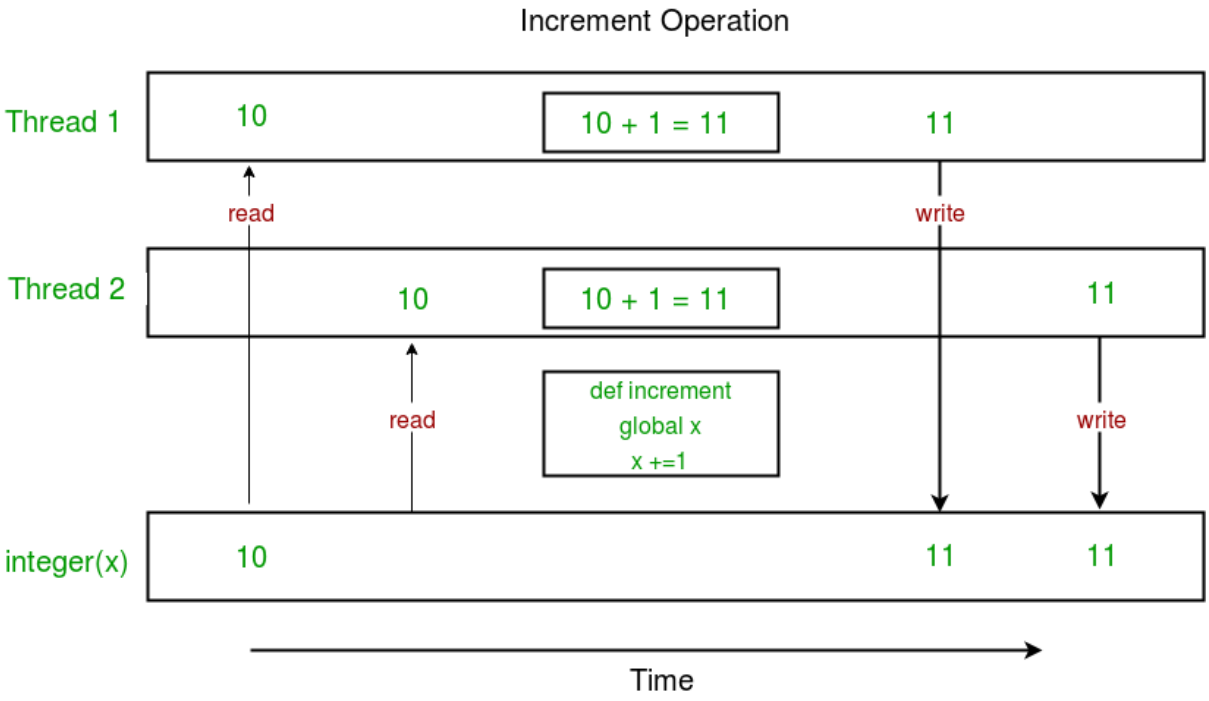
In above program:

-Two threads t1 and t2 are created in main_task function and global variable x is set to 0.

-Each thread has a target function thread_task in which increment function is called 100000 times.

-increment function will increment the global variable x by 1 in each call.

The expected final value of x is 200000 but what we get in 10 iterations of main_task function is some different values. This happens due to concurrent access of threads to the shared variable x. This unpredictability in value of x is nothing but race condition. Given below is a diagram which shows how can race condition occur in above program:



Notice that expected value of x in above diagram is 12 but due to race condition, it turns out to be 11! Hence, we need a tool for proper synchronization between multiple threads.

-Solution: threading module provides a Lock class to deal with the race conditions. Lock is implemented using a Semaphore object provided by the Operating System.

A semaphore is a synchronization object that controls access by multiple processes/threads to a common resource in a parallel programming environment. It is simply a value in a designated place in operating system (or kernel) storage that each process/thread can check and then change. Depending on the value that is found, the process/thread can use the resource or will find that it is already in use and must wait for some period before trying again. Semaphores can be binary (0 or 1) or can have additional values. Typically, a process/thread using semaphores checks the value and then, if it using the resource, changes the value to reflect this so that subsequent semaphore users will know to wait.

-See the code that solve the problem of race condition (06.py)

As you can see in the results, the final value of x comes out to be 200000 every time (which is the expected final result). Here is a diagram given below which depicts the implementation of locks in above program: