

A 𝐏𝐫𝐨𝐠𝐫𝐚𝐦 is an executable file containing a set of instructions and passively stored on disk. One program can have multiple processes. For example, the Chrome browser creates a different process for every single tab.

A 𝐏𝐫𝐨𝐜𝐞𝐬𝐬 means a program is in execution. When a program is loaded into the memory and becomes active, the program becomes a process. The process requires some essential resources such as registers, program counter, and stack.

A 𝐓𝐡𝐫𝐞𝐚𝐝 is the smallest unit of execution within a process. The following process explains the relationship between program, process, and thread.

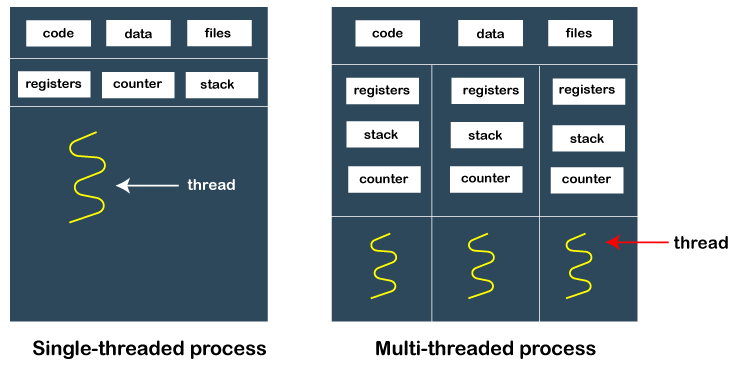
1. The program contains a set of instructions.

2. The program is loaded into memory. It becomes one or more running processes.

3. When a process starts, it is assigned memory and resources. A process can have one or more threads.

Processes are usually independent, while threads exist as subsets of a process.

* Each process has its own memory space. Threads that belong to the same process share the same memory.
* A process is a heavyweight operation. It takes more time to create and terminate.



A process can contain a single thread to multiple threads. A thread works as follows:

When a process starts, OS assigns the memory and resources to it. Each thread within a process shares the memory and resources of that process only.

Threads are mainly used to improve the processing of an application. In reality, only a single thread is executed at a time, but due to fast context switching between threads gives an illusion that threads are running parallelly.

If a single thread executes in a process, it is known as a single-threaded And if multiple threads execute simultaneously, then it is known as multithreading.

| **Parameter** | **Process** | **Thread** |
| --- | --- | --- |
| Definition | Process means a program is in execution. | Thread means a segment of a process. |
| Lightweight | The process is not Lightweight. | Threads are Lightweight. |
| Termination time | The process takes more time to terminate. | The thread takes less time to terminate. |
| Creation time | It takes more time for creation. | It takes less time for creation. |
| Communication | Communication between processes needs more time compared to thread. | Communication between threads requires less time compared to processes. |
| Context switching time | It takes more time for context switching. | It takes less time for context switching. |
| Resource | Process consume more resources. | Thread consume fewer resources. |
| Treatment by OS | Different process are tread separately by OS. | All the level peer threads are treated as a single task by OS. |
| Memory | The process is mostly isolated. | Threads share memory. |
| Sharing | It does not share data | Threads share data with each other. |

|  |  |
| --- | --- |
| Processes don’t share the memory with other processes. | Threads share the memory with other threads of the same process. |

Benefits of threading:

For a single process, multiple threads can be used to process and share the same data-space and can communicate with each other by sharing information.

They use lesser memory overhead, and hence they are called lightweight processes.

A program can remain responsive to input when threads are used.

Threads can share and process the global variable's memory.

What is Multithreading?

Multithreading refers to multiple threads of execution within an operating system. In simple terms, two or more threads of a same process are executing simultaneously.

What Are Processes?

Computer programs are merely executables, binary (or otherwise), which reside on disk. They do not take on a life of their own until loaded into memory and invoked by the operating system. A process (sometimes called a heavyweight process) is a program in execution. Each process has its own address space, memory, a data stack, and other auxiliary data to keep track of execution. The operating system manages the execution of all processes on the system, dividing the time fairly between all processes.

Processes can also fork or spawn new processes to perform other tasks, but each new process has its own memory, data stack, etc., and cannot generally share information unless inter process communication (IPC) is employed.

18.2.2. What Are Threads?

Threads (sometimes called lightweight processes) are similar to processes except that they all execute within the same process, and thus all share the same context. They can be thought of as "miniprocesses" running in parallel within a main process or "main thread."

A thread has a beginning, an execution sequence, and a conclusion. It has an instruction pointer that keeps track of where within its context it is currently running. It can be preempted (interrupted) and temporarily put on hold (also known as sleeping) while other threads are running this is called yielding.

Multiple threads within a process share the same data space with the main thread and can therefore share information or communicate with one another more easily than if they were separate processes.

Threads are generally executed in a concurrent fashion, and it is this parallelism and data sharing that enable the coordination of multiple tasks. Naturally, it is impossible to run truly in a concurrent manner in a single CPU system, so threads are scheduled in such a way that they run for a little bit, then yield to other threads (going to the proverbial "back of the line" to await more CPU time again). Throughout the execution of the entire process, each thread performs its own, separate tasks, and communicates the results with other threads as necessary.

Of course, such sharing is not without its dangers. If two or more threads access the same piece of data, inconsistent results may arise because of the ordering of data access. This is commonly known as a race condition. Fortunately, most thread libraries come with some sort of synchronization primitives that allow the thread manager to control execution and access.

Another caveat is that threads may not be given equal and fair execution time. This is because some functions block until they have completed. If not written specifically to take threads into account, this skews the amount of CPU time in favor of such greedy functions.

**Global Interpreter Lock (GIL):**

Execution of Python code is controlled by the Python Virtual Machine (aka the interpreter main loop). Python was designed in such a way that only one thread of control may be executing in this main loop, similar to how multiple processes in a system share a single CPU. Many programs may be in memory, but only one is live on the CPU at any given moment. Multiple threads may be "running" within the Python interpreter, only one thread is being executed by the interpreter at any given time.

Access to the Python Virtual Machine is controlled by the global interpreter lock (GIL). This lock is what ensures that exactly one thread is running. The Python Virtual Machine executes in the following manner in an MT environment:

1. Set the GIL

2. Switch in a thread to run

3. Execute either ...

a.For a specified number of bytecode instructions, or

b.If the thread voluntarily yields control (can be accomplished time.sleep(0))

4. Put the thread back to sleep (switch out thread)

5. Unlock the GIL, and ...

6. Do it all over again (lather, rinse, repeat)

The Python Global Interpreter Lock or GIL, in simple words, is a mutex (or a lock) that allows only one thread to hold the control of the Python interpreter.

This means that only one thread can be in a state of execution at any point in time. The impact of the GIL isn’t visible to developers who execute single-threaded programs, but it can be a performance bottleneck in CPU-bound and multi-threaded code.

Since the GIL allows only one thread to execute at a time even in a multi-threaded architecture with more than one CPU core, the GIL has gained a reputation as an “infamous” feature of Python.

CPU-bound programs are those that are pushing the CPU to its limit. This includes programs that do mathematical computations like matrix multiplications, searching, image processing, etc.

I/O-bound programs are the ones that spend time waiting for Input/Output which can come from a user, file, database, network, etc. I/O-bound programs sometimes have to wait for a significant amount of time till they get what they need from the source due to the fact that the source may need to do its own processing before the input/output is ready, for example, a user thinking about what to enter into an input prompt or a database query running in its own process.

What Problem Did the GIL Solve for Python?

Python uses reference counting for memory management. The reference counter counts the total number of references made internally in Python to assign a value to a data object. When the reference counts reaches zero, the memory occupied by the object is released.. It means that objects created in Python have a reference count variable that keeps track of the number of references that point to the object. When this count reaches zero,

example to demonstrate how reference counting works:

import sys

a = []

b = a

sys.getrefcount(a)

3

In the above example, the reference count for the empty list object [] was 3. The list object was referenced by a, b and the argument passed to sys.getrefcount().

Python Global Interpreter Lock or GIL is an important part of multithreading programming. It is a type of process lock used when working with multiple processes. It gives the control to only one thread. Generally, Python uses a single thread to run a single process. We get the same performance result of the single-threaded and multi-threaded processes using the GIL. It restricts achieving multithreading in Python because it prevents the threads and works as a single thread.

