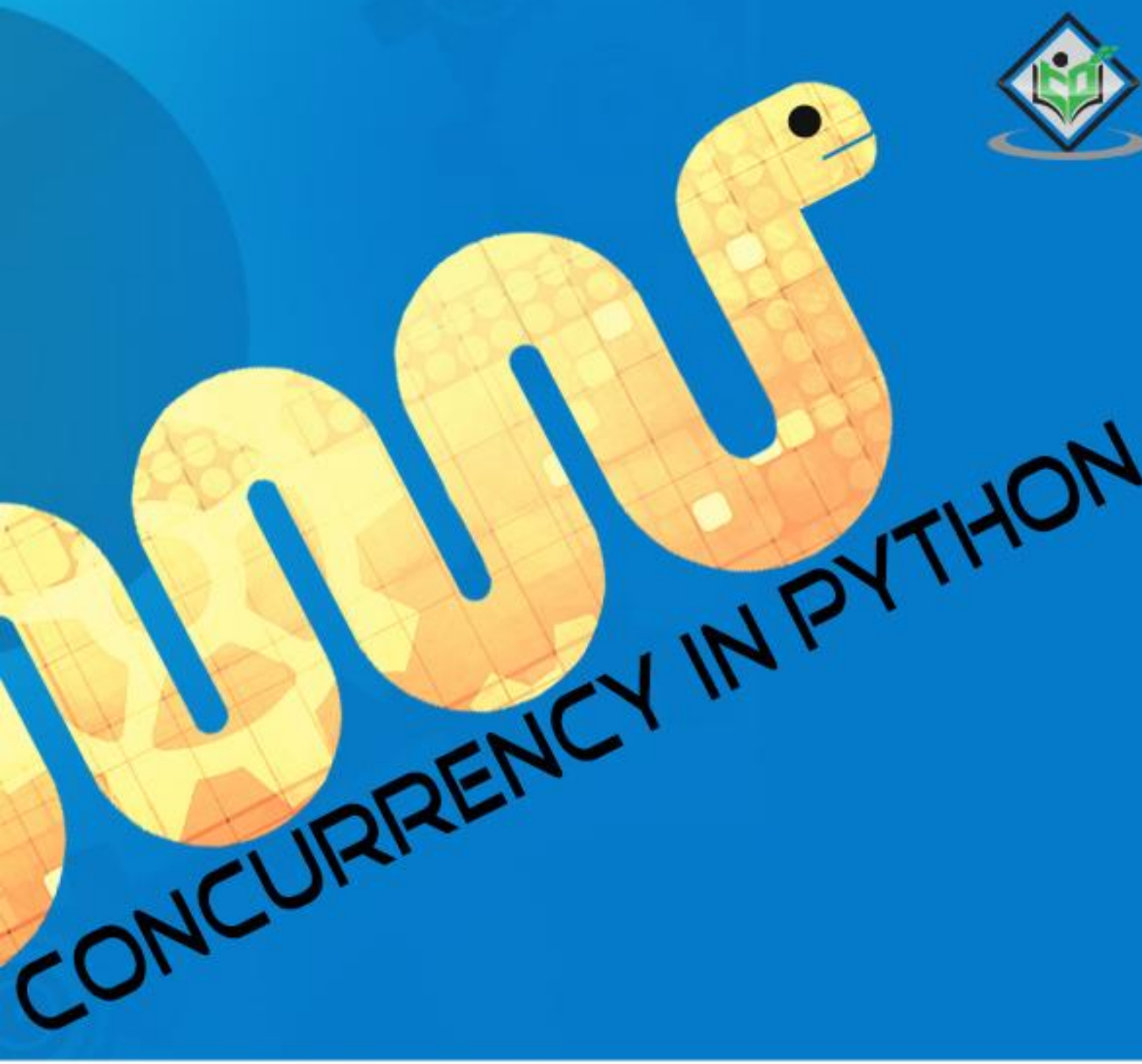


tutorialspoint

SIMPLY EASY LEARNING

www.tutorialspoint.com

https://www.facebook.com/tutorialspointindia https://twitter.com/tutorialspoint

About the Tutorial

Concurrency, natural phenomena, is the happening of two or more events at the same time. It is a challenging task for the professionals to create concurrent applications and get the most out of computer hardware.

Audience

This tutorial will be useful for graduates, postgraduates, and research students who either have an interest in this subject or have this subject as a part of their curriculum. The reader can be a beginner or an advanced learner.

Prerequisites

The reader must have basic knowledge about concepts such as Concurrency, Multiprocessing, Threads, and Process etc. of Operating System. He/she should also be aware about basic terminologies used in OS along with Python programming concepts.

Copyright & Disclaimer

Table of Contents

1. Concurrency in Python – Introduction

In this chapter, we will understand the concept of concurrency in Python and learn about the different threads and processes.

What is Concurrency?

In simple words, concurrency is the occurrence of two or more events at the same time. Concurrency is a natural phenomenon because many events occur simultaneously at any given time.

In terms of programming, concurrency is when two tasks overlap in execution. With concurrent programming, the performance of our applications and software systems can be improved because we can concurrently deal with the requests rather than waiting for a previous one to be completed.

Historical Review of Concurrency

Followings points will give us the brief historical review of concurrency:

From the concept of railroads

Concurrency is closely related with the concept of railroads. With the railroads, there was a need to handle multiple trains on the same railroad system in such a way that every train would get to its destination safely.

Concurrent computing in academia

The interest in computer science concurrency began with the research paper published by Edsger W. Dijkstra in 1965. In this paper, he identified and solved the problem of mutual exclusion, the property of concurrency control.

High-level concurrency primitives

In recent times, programmers are getting improved concurrent solutions because of the introduction of high-level concurrency primitives.

Improved concurrency with programming languages

Programming languages such as Google's Golang, Rust and Python have made incredible developments in areas which help us get better concurrent solutions.

What is thread & multithreading?

Thread is the smallest unit of execution that can be performed in an operating system. It is not itself a program but runs within a program. In other words, threads are not

independent of one other. Each thread shares code section, data section, etc. with other threads. They are also known as lightweight processes.

A thread consists of the following components:

- Program counter which consist of the address of the next executable instruction
- Stack
- Set of registers
- A unique id

Multithreading, on the other hand, is the ability of a CPU to manage the use of operating system by executing multiple threads concurrently. The main idea of multithreading is to achieve parallelism by dividing a process into multiple threads. The concept of multithreading can be understood with the help of the following example.

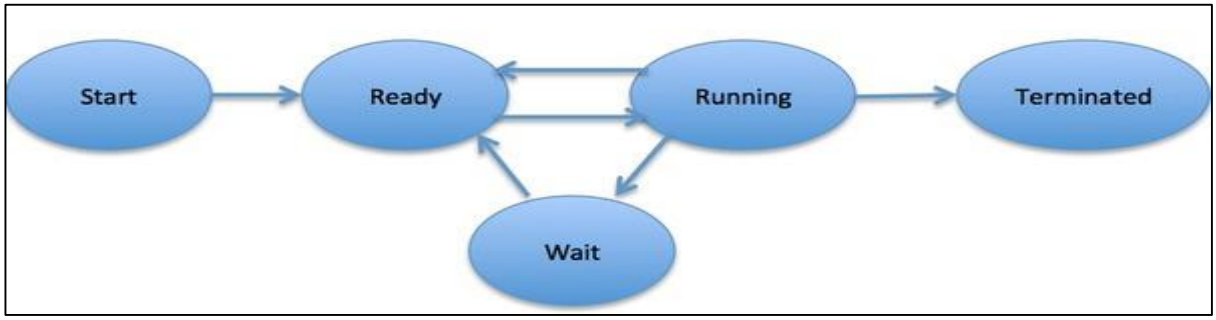
Example

Suppose we are running a particular process wherein we open MS Word to type content into it. One thread will be assigned to open MS Word and another thread will be required to type content in it. And now, if we want to edit the existing then another thread will be required to do the editing task and so on.

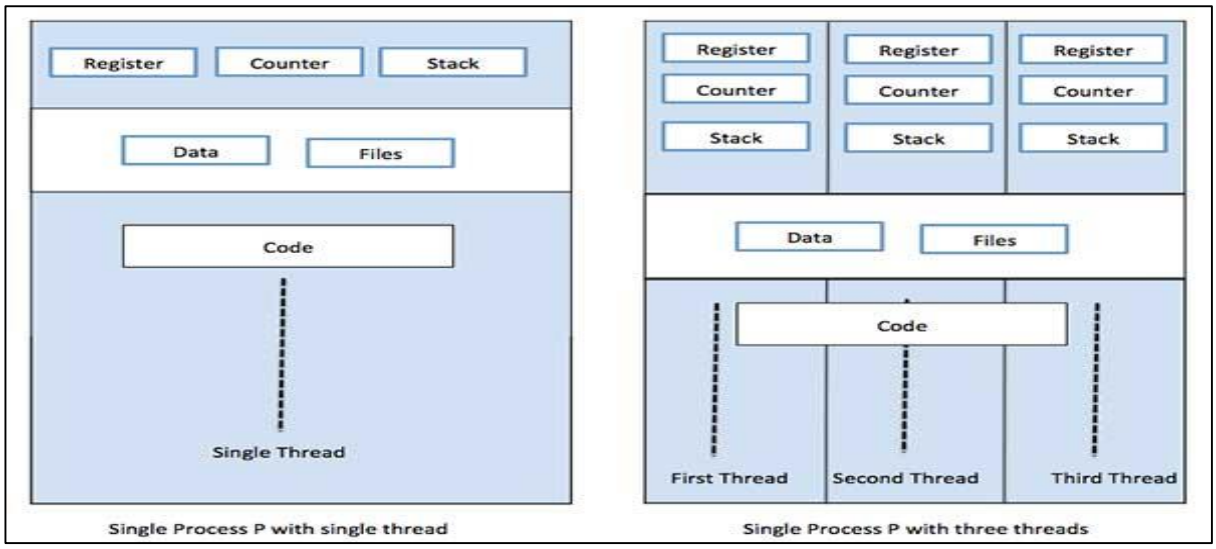
What is process & multiprocessing?

A **process** is defined as an entity, which represents the basic unit of work to be implemented in the system. To put it in simple terms, we write our computer programs in a text file and when we execute this program, it becomes a process that performs all the tasks mentioned in the program. During the process life cycle, it passes through different stages – Start, Ready, Running, Waiting and Terminating.

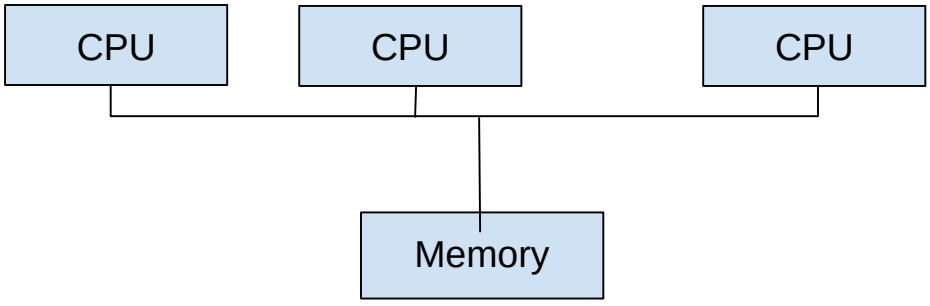
Following diagram shows the different stages of a process:



A process can have only one thread, called primary thread, or multiple threads having their own set of registers, program counter and stack. Following diagram will show us the difference:



Multiprocessing, on the other hand, is the use of two or more CPUs units within a single computer system. Our primary goal is to get the full potential from our hardware. To achieve this, we need to utilize full number of CPU cores available in our computer system. Multiprocessing is the best approach to do so.



Python is one of the most popular programming languages. Followings are some reasons that make it suitable for concurrent applications:

Syntactic sugar

Syntactic sugar is syntax within a programming language that is designed to make things easier to read or to express. It makes the language "sweeter" for human use: things can be expressed more clearly, more concisely, or in an alternative style based on preference. Python comes with Magic methods, which can be defined to act on objects. These Magic methods are used as syntactic sugar and bound to more easy-to-understand keywords.

Large Community

Python language has witnessed a massive adoption rate amongst data scientists and mathematicians, working in the field of AI, machine learning, deep learning and quantitative analysis.

Useful APIs for concurrent programming

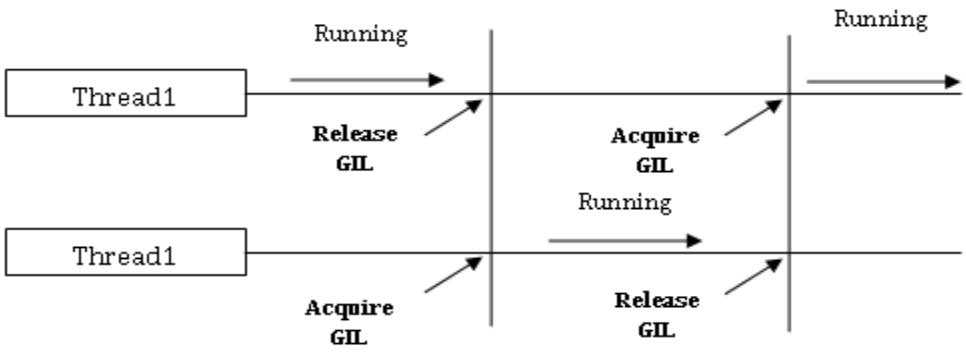
Python 2 and 3 have large number of APIs dedicated for parallel/concurrent programming. Most popular of them are **threading**, **concurrent.features**, **multiprocessing**, a**syncio**, **gevent and greenlets,** etc.

Limitations of Python in implementing concurrent applications

Python comes with a limitation for concurrent applications. This limitation is called **GIL (Global Interpreter Lock)** is present within Python. GIL never allows us to utilize multiple cores of CPU and hence we can say that there are no true threads in Python. We can understand the concept of GIL as follows:

GIL (Global Interpreter Lock)

It is one of the most controversial topics in the Python world. In CPython, GIL is the mutex - the mutual exclusion lock, which makes things thread safe. In other words, we can say that GIL prevents multiple threads from executing Python code in parallel. The lock can be held by only one thread at a time and if we want to execute a thread then it must acquire the lock first. The diagram shown below will help you understand the working of GIL.



However, there are some libraries and implementations in Python such as **Numpy**, **Jpython** and **IronPython.** These libraries work without any interaction with GIL.

2. Concurrency in Python – Concurrency vs Parallelism

Both concurrency and parallelism are used in relation to multithreaded programs but there is a lot of confusion about the similarity and difference between them. The big question in this regard: **is concurrency parallelism or not?** Although both the terms appear quite similar but the answer to the above question is NO, concurrency and parallelism are not same. Now, if they are not same then what is the basic difference between them?

In simple terms, concurrency deals with managing the access to shared state from different threads and on the other side, parallelism deals with utilizing multiple CPUs or its cores to improve the performance of hardware.

Concurrency in Detail

Concurrency is when two tasks overlap in execution. It could be a situation where an application is progressing on more than one task at the same time. We can understand it diagrammatically; multiple tasks are making progress at the same time, as follows:

Levels of Concurrency

In this section, we will discuss the three important levels of concurrency in terms of programming:

Low-Level Concurrency

In this level of concurrency, there is explicit use of atomic operations. We cannot use such kind of concurrency for application building, as it is very error-prone and difficult to debug. Even Python does not support such kind of concurrency.

Mid-Level Concurrency

In this concurrency, there is no use of explicit atomic operations. It uses the explicit locks. Python and other programming languages support such kind of concurrency. Mostly application programmers use this concurrency.

High-Level Concurrency

In this concurrency, neither explicit atomic operations nor explicit locks are used. Python has **concurrent.futures** module to support such kind of concurrency.

Properties of Concurrent Systems

For a program or concurrent system to be correct, some properties must be satisfied by it. Properties related to the termination of system are as follows:

Correctness property

The correctness property means that the program or the system must provide the desired correct answer. To keep it simple, we can say that the system must map the starting program state to final state correctly.

Safety property

The safety property means that the program or the system must remain in a **"good"** or **"safe"** state and never does anything **"bad"**.

Liveness property

This property means that a program or system must **"make progress"** and it would reach at some desirable state.

Actors of concurrent systems

This is one common property of concurrent system in which there can be multiple processes and threads, which run at the same time to make progress on their own tasks. These processes and threads are called actors of the concurrent system.

Resources of Concurrent Systems

The actors must utilize the resources such as memory, disk, printer etc. in order to perform their tasks.

Certain set of rules

Every concurrent system must possess a set of rules to define the kind of tasks to be performed by the actors and the timing for each. The tasks could be acquiring of locks, memory sharing, modifying the state, etc.

Barriers of Concurrent Systems

While implementing concurrent systems, the programmer must take into consideration the following two important issues, which can be the barriers of concurrent systems:

Sharing of data

An important issue while implementing the concurrent systems is the sharing of data among multiple threads or processes. Actually, the programmer must ensure that locks protect the shared data so that all the accesses to it are serialized and only one thread or process can access the shared data at a time. In case, when multiple threads or processes are all trying to access the same shared data then not all but at least one of them would be blocked and would remain idle. In other words, we can say that we would be able to use only one process or thread at a time when lock is in force. There can be some simple solutions to remove the above-mentioned barriers:

Data Sharing Restriction

The simplest solution is not to share any mutable data. In this case, we need not to use explicit locking and the barrier of concurrency due to mutual data would be solved.

Data Structure Assistance

Many times the concurrent processes need to access the same data at the same time. Another solution, than using of explicit locks, is to use a data structure that supports concurrent access. For example, we can use the **queue** module, which provides thread-safe queues. We can also use **multiprocessing.JoinableQueue** classes for multiprocessing-based concurrency.

Immutable Data Transfer

Sometimes, the data structure that we are using, say concurrency queue, is not suitable then we can pass the immutable data without locking it.

Mutable Data Transfer

In continuation of the above solution, suppose if it is required to pass only mutable data, rather than immutable data, then we can pass mutable data that is read only.

Sharing of I/O Resources

Another important issue in implementing concurrent systems is the use of I/O resources by threads or processes. The problem arises when one thread or process is using the I/O for such a long time and other is sitting idle. We can see such kind of barrier while working with an I/O heavy application. It can be understood with the help of an example, the requesting of pages from web browser. It is a heavy application. Here, if the rate at which the data is requested is slower than the rate at which it is consumed then we have I/O barrier in our concurrent system.

The following Python script is for requesting a web page and getting the time our network took to get the requested page:

```
import urllib.request


import time

```

```
ts = time.time()

req = urllib.request.urlopen('http://www.tutorialspoint.com')

pageHtml = req.read()

te = time.time()

print("Page Fetching Time : {} Seconds".format (te-ts))
```

After executing the above script, we can get the page fetching time as shown below.

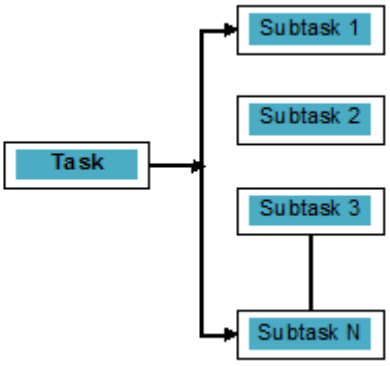
Output

```
Page Fetching Time: 1.0991398811340332 Seconds
```

We can see that the time to fetch the page is more than one second. Now what if we want to fetch thousands of different web pages, you can understand how much time our network would take.

What is Parallelism?

Parallelism may be defined as the art of splitting the tasks into subtasks that can be processed simultaneously. It is opposite to the concurrency, as discussed above, in which two or more events are happening at the same time. We can understand it diagrammatically; a task is broken into a number of subtasks that can be processed in parallel, as follows:



To get more idea about the distinction between concurrency and parallelism, consider the following points:

Concurrent but not parallel

An application can be concurrent but not parallel means that it processes more than one task at the same time but the tasks are not broken down into subtasks.

Parallel but not concurrent

An application can be parallel but not concurrent means that it only works on one task at a time and the tasks broken down into subtasks can be processed in parallel.

Neither parallel nor concurrent

An application can be neither parallel nor concurrent. This means that it works on only one task at a time and the task is never broken into subtasks.

Both parallel and concurrent

An application can be both parallel and concurrent means that it both works on multiple tasks at a time and the task is broken into subtasks for executing them in parallel.

Necessity of Parallelism

We can achieve parallelism by distributing the subtasks among different cores of single CPU or among multiple computers connected within a network.

Consider the following important points to understand why it is necessary to achieve parallelism:

Efficient code execution

With the help of parallelism, we can run our code efficiently. It will save our time because the same code in parts is running in parallel.

Faster than sequential computing

Sequential computing is constrained by physical and practical factors due to which it is not possible to get faster computing results. On the other hand, this issue is solved by parallel computing and gives us faster computing results than sequential computing.

Less execution time

Parallel processing reduces the execution time of program code.

If we talk about real life example of parallelism, the graphics card of our computer is the example that highlights the true power of parallel processing because it has hundreds of individual processing cores that work independently and can do the execution at the same time. Due to this reason, we are able to run high-end applications and games as well.

Understanding of the processors for implementation

We know about concurrency, parallelism and the difference between them but what about the system on which it is to be implemented. It is very necessary to have the understanding of the system, on which we are going to implement, because it gives us the

benefit to take informed decision while designing the software. We have the following two kinds of processors:

Single-core processors

Single-core processors are capable of executing one thread at any given time. These processors use **context switching** to store all the necessary information for a thread at a specific time and then restoring the information later. The context switching mechanism helps us make progress on a number of threads within a given second and it looks as if the system is working on multiple things.

Single-core processors come with many advantages. These processors require less power and there is no complex communication protocol between multiple cores. On the other hand, the speed of single-core processors is limited and it is not suitable for larger applications.

Multi-core processors

Multi-core processors have multiple independent processing units also called **cores.**

Such processors do not need context switching mechanism as each core contains everything it needs to execute a sequence of stored instructions.

Fetch-Decode-Execute Cycle

The cores of multi-core processors follow a cycle for executing. This cycle is called the **Fetch-Decode-Execute** cycle. It involves the following steps:

Fetch

This is the first step of cycle, which involves the fetching of instructions from the program memory.

Decode

Recently fetched instructions would be converted to a series of signals that will trigger other parts of the CPU.

Execute

It is the final step in which the fetched and the decoded instructions would be executed. The result of execution will be stored in a CPU register.

One advantage over here is that the execution in multi-core processors are faster than that of single-core processors. It is suitable for larger applications. On the other hand, complex communication protocol between multiple cores is an issue. Multiple cores require more power than single-core processors.

3. Concurrency in Python – System & Memory Architecture

There are different system and memory architecture styles that need to be considered while designing the program or concurrent system. It is very necessary because one system & memory style may be suitable for one task but may be error prone to other task.

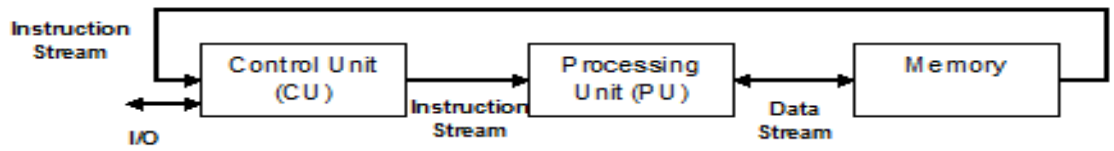
Computer system architectures supporting concurrency

Michael Flynn in 1972 gave taxonomy for categorizing different styles of computer system architecture. This taxonomy defines four different styles as follows:

- Single instruction stream, single data stream (SISD)
- Single instruction stream, multiple data stream (SIMD)
- Multiple instruction stream, single data stream (MISD)
- Multiple instruction stream, multiple data stream (MIMD).

Single instruction stream, single data stream (SISD)

As the name suggests, such kind of systems would have one sequential incoming data stream and one single processing unit to execute the data stream. They are just like uniprocessor systems having parallel computing architecture. Following is the architecture of SISD:



Advantages of SISD

The advantages of SISD architecture are as follows:

- It requires less power.
- There is no issue of complex communication protocol between multiple cores.

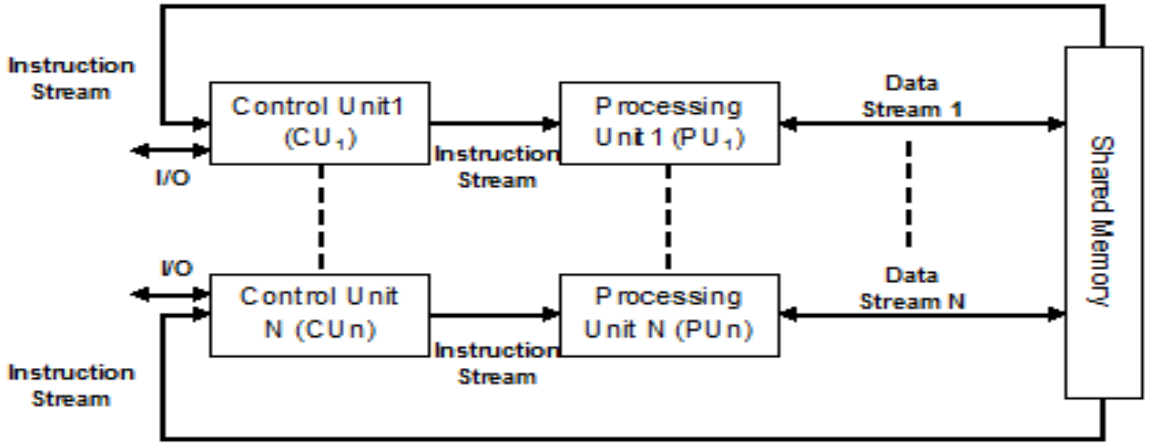
Disadvantages of SISD

The disadvantages of SISD architecture are as follows:

- The speed of SISD architecture is limited just like single-core processors.
- It is not suitable for larger applications.

Single instruction stream, multiple data stream (SIMD)

As the name suggests, such kind of systems would have multiple incoming data streams and number of processing units that can act on a single instruction at any given time. They are just like multiprocessor systems having parallel computing architecture. Following is the architecture of SIMD:



The best example for SIMD is the graphics cards. These cards have hundreds of individual processing units. If we talk about computational difference between SISD and SIMD then for the adding arrays **[5, 15, 20]** and **[15, 25, 10],** SISD architecture would have to perform three different add operations. On the other hand, with the SIMD architecture, we can add then in a single add operation.

Advantages of SIMD

The advantages of SIMD architecture are as follows:

- Same operation on multiple elements can be performed using one instruction only.
- Throughput of the system can be increased by increasing the number of cores of the processor.
- Processing speed is higher than SISD architecture.

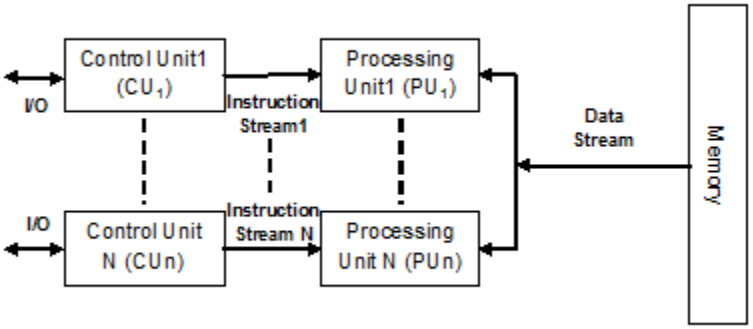
Disadvantages of SIMD

The disadvantages of SIMD architecture are as follows:

- There is complex communication between numbers of cores of processor.
- The cost is higher than SISD architecture.

Multiple Instruction Single Data (MISD) stream

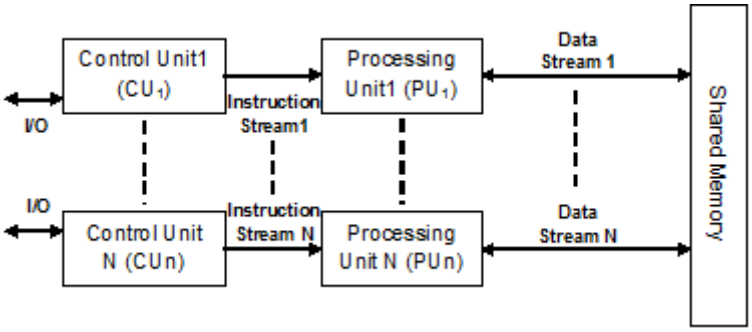
Systems with MISD stream have number of processing units performing different operations by executing different instructions on the same data set. Following is the architecture of MISD:



The representatives of MISD architecture do not yet exist commercially.

Multiple Instruction Multiple Data (MIMD) stream

In the system using MIMD architecture, each processor in a multiprocessor system can execute different sets of instructions independently on the different set of data set in parallel. It is opposite to SIMD architecture in which single operation is executed on multiple data sets. Following is the architecture of MIMD:



A normal multiprocessor uses the MIMD architecture. These architectures are basically used in a number of application areas such as computer-aided design/computer-aided manufacturing, simulation, modeling, communication switches, etc.

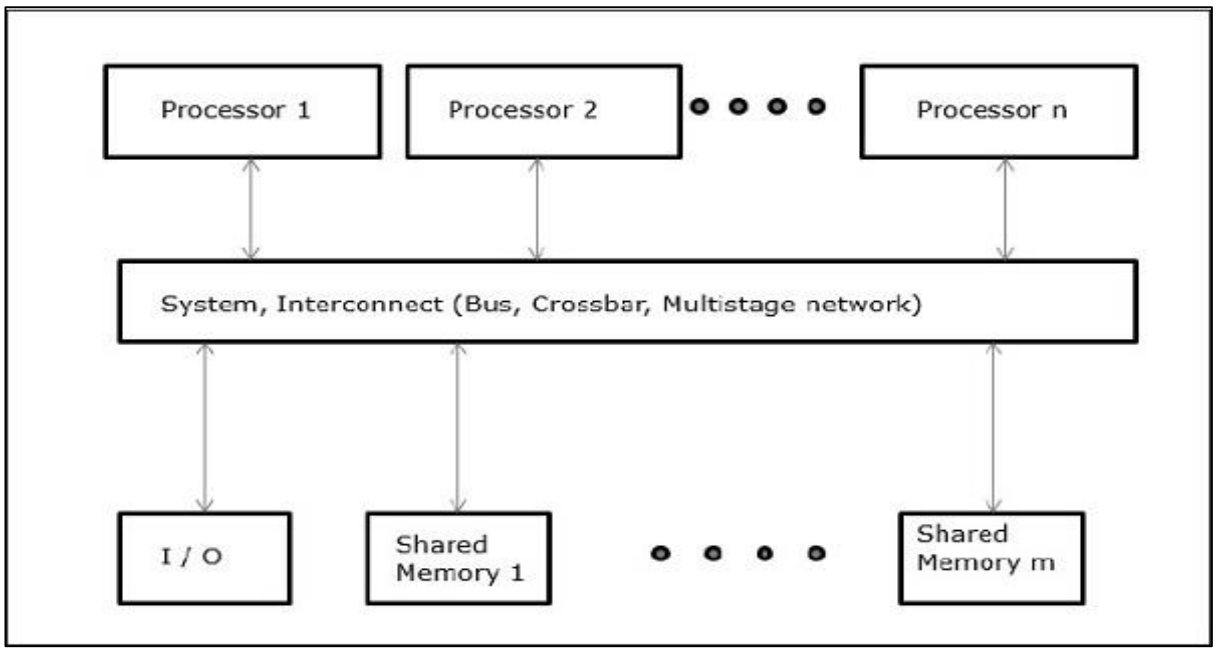
Memory architectures supporting concurrency

While working with the concepts like concurrency and parallelism, there is always a need to speed up the programs. One solution found by computer designers is to create shared-memory multi-computers, i.e., computers having single physical address space, which is accessed by all the cores that a processor is having. In this scenario, there can be a number of different styles of architecture but following are the three important architecture styles:

UMA (Uniform Memory Access)

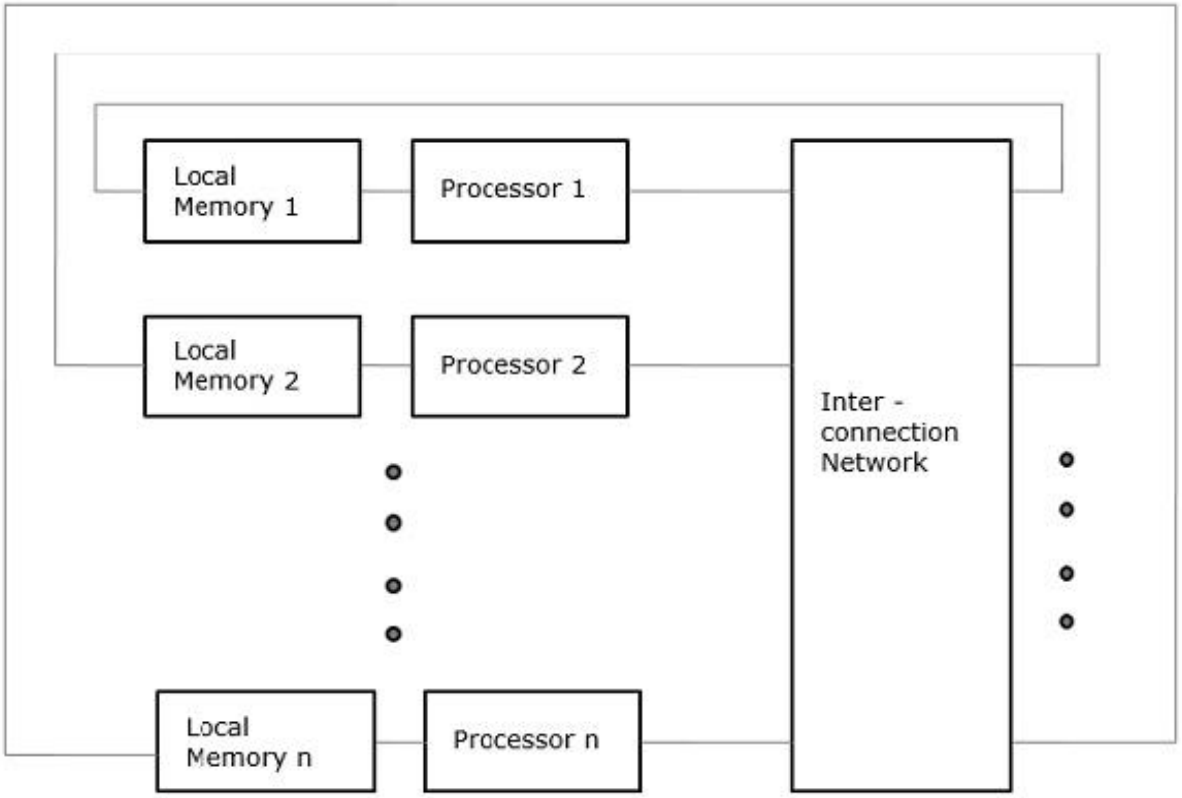
In this model, all the processors share the physical memory uniformly. All the processors have equal access time to all the memory words. Each processor may have a private cache memory. The peripheral devices follow a set of rules.

When all the processors have equal access to all the peripheral devices, the system is called a **symmetric multiprocessor**. When only one or a few processors can access the peripheral devices, the system is called an **asymmetric multiprocessor**.



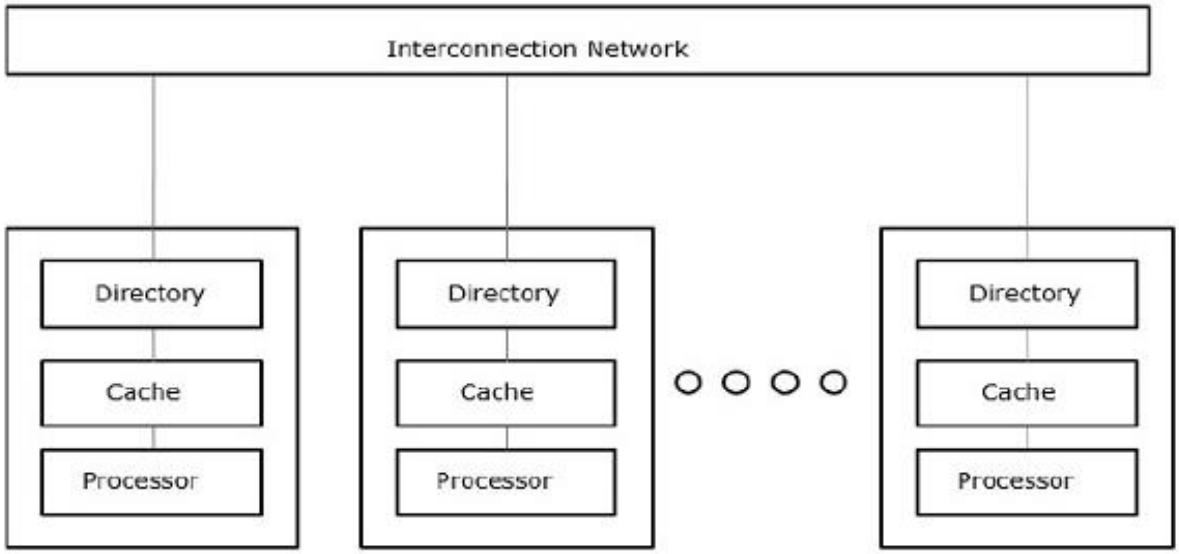
Non-uniform Memory Access (NUMA)

In the NUMA multiprocessor model, the access time varies with the location of the memory word. Here, the shared memory is physically distributed among all the processors, called local memories. The collection of all local memories forms a global address space which can be accessed by all the processors.



Cache Only Memory Architecture (COMA)

The COMA model is a specialized version of the NUMA model. Here, all the distributed main memories are converted to cache memories.



4. Concurrency in Python – Threads

In general, as we know that thread is a very thin twisted string usually of the cotton or silk fabric and used for sewing clothes and such. The same term thread is also used in the world of computer programming. Now, how do we relate the thread used for sewing clothes and the thread used for computer programming? The roles performed by the two threads is similar here. In clothes, thread hold the cloth together and on the other side, in computer programming, thread hold the computer program and allow the program to execute sequential actions or many actions at once.

Thread is the smallest unit of execution in an operating system. It is not in itself a program but runs within a program. In other words, threads are not independent of one other and share code section, data section, etc. with other threads. These threads are also known as lightweight processes.

States of Thread

To understand the functionality of threads in depth, we need to learn about the lifecycle of the threads or the different thread states. Typically, a thread can exist in five distinct states. The different states are shown below:

New Thread

A new thread begins its life cycle in the new state. However, at this stage, it has not yet started and it has not been allocated any resources. We can say that it is just an instance of an object.

Runnable

As the newly born thread is started, the thread becomes runnable i.e. waiting to run. In this state, it has all the resources but still task scheduler have not scheduled it to run.

Running

In this state, the thread makes progress and executes the task, which has been chosen by task scheduler to run. Now, the thread can go to either the dead state or the non-runnable/ waiting state.

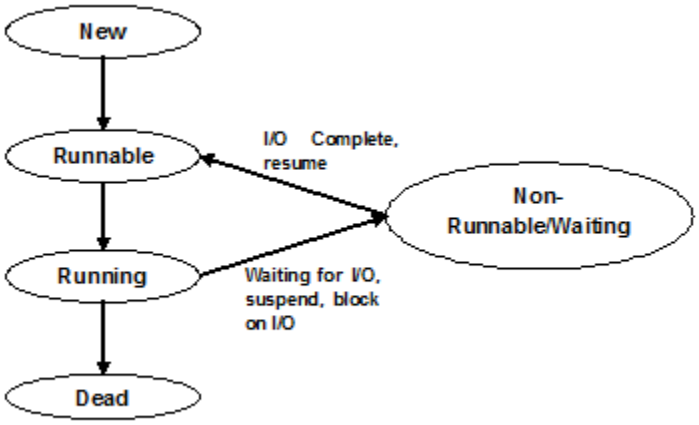
Non-running/waiting

In this state, the thread is paused because it is either waiting for the response of some I/O request or waiting for the completion of the execution of other thread.

Dead

A runnable thread enters the terminated state when it completes its task or otherwise terminates.

The following diagram shows the complete life cycle of a thread:



Types of Thread

In this section, we will see the different types of thread. The types are described below:

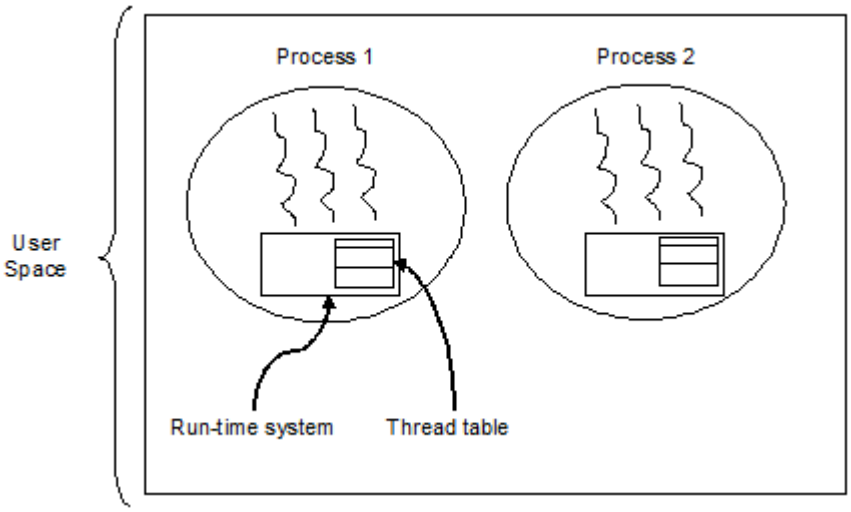
User Level Threads

These are user-managed threads.

In this case, the thread management kernel is not aware of the existence of threads. The thread library contains code for creating and destroying threads, for passing message and data between threads, for scheduling thread execution and for saving and restoring thread contexts. The application starts with a single thread.

The examples of user level threads are:

- Java threads
- POSIX threads



Advantages of User Level Threads

Following are the different advantages of user level threads:

- Thread switching does not require Kernel mode privileges.
- User level thread can run on any operating system.
- Scheduling can be application specific in the user level thread.
- User level threads are fast to create and manage.

Disadvantages of User Level Threads

Following are the different disadvantages of user level threads:

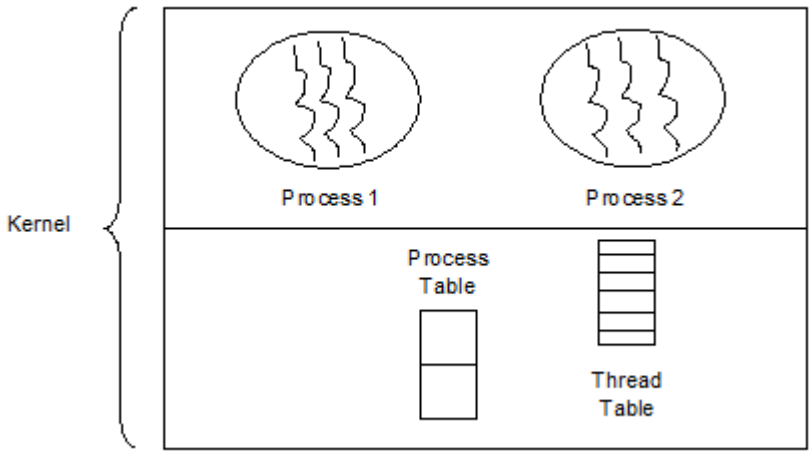
- In a typical operating system, most system calls are blocking.
- Multithreaded application cannot take advantage of multiprocessing.

Kernel Level Threads

Operating System managed threads act on kernel, which is an operating system core.

In this case, the Kernel does thread management. There is no thread management code in the application area. Kernel threads are supported directly by the operating system. Any application can be programmed to be multithreaded. All of the threads within an application are supported within a single process.

The Kernel maintains context information for the process as a whole and for individual threads within the process. Scheduling by the Kernel is done on a thread basis. The Kernel performs thread creation, scheduling and management in Kernel space. Kernel threads are generally slower to create and manage than the user threads. The examples of kernel level threads are Windows, Solaris.



Advantages of Kernel Level Threads

Following are the different advantages of kernel level threads:

- Kernel can simultaneously schedule multiple threads from the same process on multiple processes.
- If one thread in a process is blocked, the Kernel can schedule another thread of the same process.
- Kernel routines themselves can be multithreaded.

Disadvantages of Kernel Level Threads

- Kernel threads are generally slower to create and manage than the user threads.
- Transfer of control from one thread to another within the same process requires a mode switch to the Kernel.

Thread Control Block - TCB

Thread Control Block (TCB) may be defined as the data structure in the kernel of operating system that mainly contains information about thread. Thread-specific information stored in TCB would highlight some important information about each process.

Consider the following points related to the threads contained in TCB:

- **Thread identification:** It is the unique thread id (tid) assigned to every new thread.
- **Thread state:** It contains the information related to the state (Running, Runnable, Non-Running, Dead) of the thread.
- **Program Counter (PC):** It points to the current program instruction of the thread.
- **Register set:** It contains the thread's register values assigned to them for computations.
- **Stack Pointer:** It points to the thread's stack in the process. It contains the local variables under thread's scope.
- **Pointer to PCB:** It contains the pointer to the process that created that thread.

Thread identification
Thread state
Program Counter (PC)
Register set

Stack Pointer
Pointer to PCB

Relation between process & thread

In multithreading, process and thread are two very closely related terms having the same goal to make computer able to do more than one thing at a time. A process can contain one or more threads but on the contrary, thread cannot contain a process. However, they both remain the two basic units of execution. A program, executing a series of instructions, initiates process and thread both.

The following table shows the comparison between process and thread:

S.No.	Process	Thread
1	Process is heavy weight or resource intensive.	Thread is lightweight which takes fewer resources than a process.
2	Process switching needs interaction with operating system.	Thread switching does not need to interact with operating system.
3	In multiple processing environments, each process executes the same code but has its own memory and file resources.	All threads can share same set of open files, child processes.
4	If one process is blocked, then no other process can execute until the first process is unblocked.	While one thread is blocked and waiting, a second thread in the same task can run.
5	Multiple processes without using threads use more resources.	Multiple threaded processes use fewer resources.

6	In multiple processes, each process operates independently of the others.	One thread can read, write or change another thread's data.
7	If there would be any change in the parent process then it does not affect the child processes.	If there would be any change in the main thread then it may affect the behavior of other threads of that process.
8	To communicate with sibling processes, processes must use inter-process communication.	Threads can directly communicate with other threads of that process.

End of ebook preview

If you liked what you saw…

Buy it from our store @ https://store.tutorialspoint.com