1. **Многопроцессность, многопоточность, асинхронность.**

**Конкурентность (concurrency) -** это наиболее общий термин, который говорит, что одновременно выполняется более одной задачи.

**Многопроцессность (параллельное исполнение).** Процесс – контейнер, содержащий ресурсы программы (адресное пространство, потоки, дочерние процессы).

В многопроцессорной системе процессы выполняются **параллельно** (в них присутствуют несколько системных конвейеров для исполнения команд).

В однопроцессорной системе процессы выполняются **псевдопараллельно (конкурентно)**, **т.е. процессы выполняются последовательно, занимая малые кванты процессорного времени**.

**Многопоточность** - это один из способов реализации конкурентного исполнения, при котором задача разбивается на множество потоков, занимающихся её решением.

Процесс, порождённый в операционной системе, может состоять из нескольких потоков, выполняющихся **без предписанного порядка во времени**. Контекст переключается между потоками конкурентно.

Все потоки выполняются в адресном пространстве процесса. Выполняющийся процесс имеет как минимум один (главный) поток.

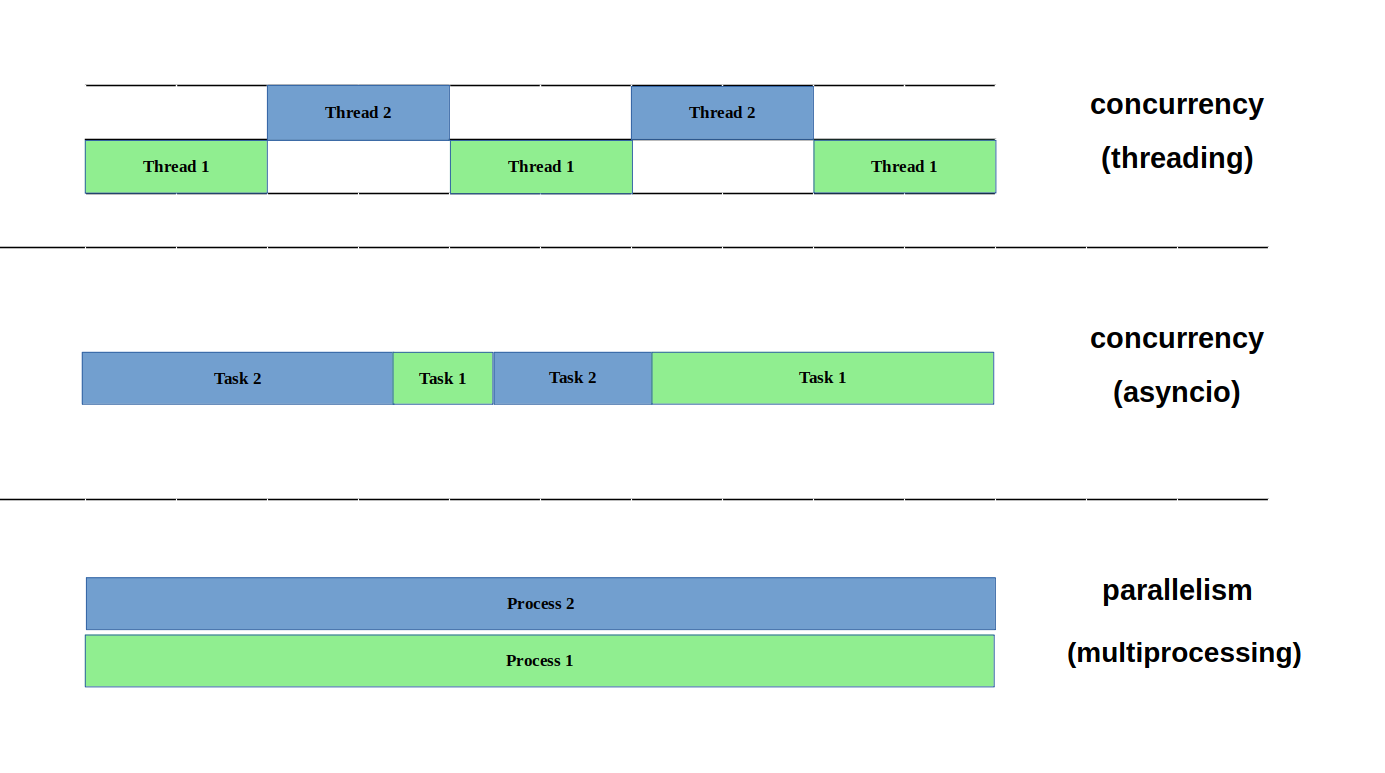
У каждого потока есть приоритет. Если есть несколько спящих потоков, которые нужно запустить, то ОС сначала запустит поток с более высоким приоритетом. Потоки с одинаковыми приоритетами запускаются в порядке очереди.

Основными проблемами при работе с многопоточностью являются **состояние гонки** и **взаимная блокировка**. Состояние гонки возникает, когда два или более потоков пытаются одновременно изменить общие данные, что может привести к непредсказуемым результатам. Взаимная блокировка — это ситуация, когда два потока блокируют друг друга, ожидая освобождения ресурсов, что приводит к "заморозке" программы. Для решения этих проблем используются различные методы синхронизации, такие как мьютексы и семафоры. В python GIL (Global Interpreter Lock).

Многопоточность прекрасно решает задачи, связанные с вводом/выводом (I/O) информации.

**Асинхронность**. Асинхронный код построен вокруг одного потока и одного процесса: он использует кооперативную многозадачность (тип многозадачности, при котором следующая задача выполняется только после того, как текущая задача явно объявит себя готовой отдать процессорное время другим задачам). Асинхронный код выполняется конкурентно.

Асинхронные подпрограммы способны вставать на паузу, ожидая получение какого-то результата и давая работать другим подпрограммам. Асинхронный код позволяет выполнять задачи одновременно. Если быть точным, то он имитирует ощущение одновременности.



1. **GIL (Global Interpreter Lock).**

Представляет собой мьютекс, который блокирует доступ к объекту Python interpreter в многопоточных средах, разрешая выполнять лишь одну инструкцию за раз.

GIL был введен не как намеренное ограничение, а скорее, как необходимая мера для обеспечения безопасности в среде многозадачности.

Один из наиболее эффективных способов обойти GIL - это использование многопроцессорной обработки вместо многозадачных потоков. Поскольку каждый процесс имеет свой собственный интерпретатор Python и собственный GIL, они могут параллельно выполняться на разных ядрах процессора.

1. **SOLID принципы**

**Single Responsibility Principle** — принцип единственной ответственности. Каждый класс должен отвечать только за одну зону ответственности (действий).

**Open Closed Principle** — принцип открытости-закрытости. Класс должен быть закрыт для изменения, но открыт для расширения (при наследовании). Это когда класс спроектирован так, что можно без труда поменять логику в унаследованном классе, не трогая базовый класс.

**Liskov Substitution Principle** — принцип подстановки Барбары Лисков. Код, который работает с базовым классом, должен работать и с любым наследуемым его классом, не вызывая ошибок или неожиданного поведения.

Дочерний класс дополняет, но не заменяет поведение базового класса. То есть в любом месте программы замена базового класса на дочерний не должна вызывать проблем.

Для того чтобы следовать принципу подстановки Барбары Лисков необходимо в базовый (родительский) класс выносить только общую логику, характерную для классов наследников.

**Interface Segregation Principle** — принцип разделения интерфейса. Клиенты не должны зависеть от интерфейсов, которые они не используют. Большие интерфейсы следует разбивать на интерфейсы поменьше. Так клиенты смогут использовать только те интерфейсы, которые им нужны.

**Dependency Inversion Principle** — принцип инверсии зависимостей. Модули верхнего уровня не должны зависеть от модулей нижнего уровня напрямую (между ними должна находится абстракция). Оба типа модулей должны зависеть от абстракций. Абстракции не должны зависеть от реализаций, но реализации должны зависеть от абстракций.

|  |  |
| --- | --- |
| Нарушение принципа:  Зависимость StudentService от StudentRepository определена внутри класса. | Зависимость StudentRepository передается в StudentService через конструктор. Интерфейс StudentRepository позволяет избежать конкретной реализации хранилища студентов в классе StudentService. Вместо этого реализацию базы можно сделать в классе DatabaseStudentRepository без изменения кода StudentService. |
| class StudentRepository:  def save(self, student):  pass  def findById(self, id):  return None  class StudentService:  def \_\_init\_\_(self):  self.repository = StudentRepository()  def add\_student(self, student):  self.repository.save(student)  def find\_student(self, id):  return self.repository.findById(id) | class StudentRepository:  def save(self, student):  pass  def findById(self, id):  return None  class StudentService:  def \_\_init\_\_(self, repository):  self.repository = repository  def add\_student(self, student):  self.repository.save(student)  def find\_student(self, id):  return self.repository.findById(id) |