Learning Multithreading & Multiprogramming

- **What is a Thread?**
  - A **thread** is the smallest unit of execution inside a program (process).
  - Multiple threads can exist within a single process and share the same memory.
  - Threads are lightweight and are best for tasks that mostly **wait** (I/O operations, network requests, file reading, etc.).

- **What is Multithreading?**
  - **Multithreading** is running multiple threads at the same time within a single process.
  - In Python, because of the **Global Interpreter Lock (GIL)**, threads cannot truly run Python bytecode in parallel, but they are great for I/O-bound tasks.
  - Example use cases:
    - Handling multiple client requests in a server.
    - Downloading multiple files at once.
    - Reading/writing files while performing other operations.

- **What is a Process?**
  - A **process** is an independent program with its own memory space.
  - Unlike threads, processes do **not share memory** (each has its own copy).
  - Processes are heavier than threads but allow true parallel execution on multiple CPU cores.

- **What is Multiprocessing?**
  - **Multiprocessing** is running multiple processes in parallel, each with its own memory and Python interpreter.
  - This bypasses the **GIL** and achieves true parallelism for CPU-heavy tasks.
  - Example use cases:
    - Machine learning model training.
    - Image and video processing.

- Data crunching and computations (sorting, math-heavy workloads).

Which Should I Use?

Situation	Best Choice	Why?
I/O-bound tasks (network requests, disk I/O, APIs, DB queries)	Multithreading	Threads can overlap waiting time efficiently.
CPU-bound tasks (math, ML, image processing, encryption, heavy loops)	Multiprocessing	Processes run in parallel on multiple CPU cores.
Lightweight background tasks (timers, monitoring, logging)	Threads	Cheaper than processes, simple to use.
High-performance parallel computing	Processes / Multiprocessing	Avoids GIL, achieves real parallelism.
IoT / Embedded devices (sensor data collection, communication with cloud, lightweight control loops)	Multithreading	Most IoT workloads are I/O-heavy (reading sensors, sending data). Threads are efficient and lightweight. Use multiprocessing only if doing CPU-heavy edge AI (like image recognition on device).

Key Takeaways

- Use threads for I/O-bound tasks where you spend time waiting.
- Use processes for CPU-bound tasks where you need true parallel computation.
- Threads share memory (faster communication, risk of race conditions).
- Processes have separate memory (safer, but higher overhead).
- In real-world systems, a combination of both is often used.
- For IoT, multithreading is usually enough unless you are doing edge AI.