*"Imagine you are at a restaurant with only* ***one waiter*** *serving all tables. Everyone has to wait their turn to order, get their food, and pay the bill. This makes everything* ***slow****."*

**How can we fix this?**  
Hire **multiple waiters** to serve different customers **at the same time**!

This is exactly what **concurrency** does in a computer program—it allows multiple tasks (threads) to run **at the same time**, making things **faster and more efficient**.

**Step 2: How This Problem Exists in Computers?**

Imagine a computer running **a program with multiple tasks**:

* **Downloading a file** from the internet
* **Processing user input** (typing in a text editor)
* **Playing music in the background**

What happens if the computer does all of these **one by one**?  
Slow performance  
Bad user experience

**Solution** → Use **concurrency** to run multiple tasks **simultaneously**!

**Data races** (two threads modifying the same data at the same time)

**What is Fearless Concurrency?**

Rust’s **fearless concurrency** means you can write **multi-threaded programs** **without worrying** about:  
- Data races  
- Deadlocks  
- Undefined behavior

Rust’s **ownership system** makes sure that **only one thread** can modify data at a time, or safely share it between multiple threads.

**Why Use Concurrency?**

- Speed up programs by doing **multiple things at once** (e.g., download while processing).  
- Efficiently use **multiple CPU cores** .  
- Handle multiple **user requests** at the same time (like in web servers).

**Where Do We Use Concurrency?**

**Web servers** (handling multiple users).  
**Game development** (physics, rendering, AI all running in parallel).  
**Database systems** (reading and writing at the same time).  
**Background tasks** (like file downloads).

**How to Use Concurrency in Rust?**

Rust provides multiple ways to handle concurrency safely:  
1️) **Threads (std::thread)** → Basic multi-threading  
2️) **Message Passing (mpsc)** → Sending data between threads  
3️) **Shared State (Arc<Mutex<T>>)** → Sharing data safely

**1️) Creating Threads in Rust**

**Example: Two friends painting a house**

Imagine two painters **working at the same time**:

* **Thread 1:** Paints the **walls**
* **Thread 2:** Paints the **roof**

**What Happens?**

* Two tasks run **at the same time**!
* The **main thread** paints the roof.
* The **new thread** paints the wall.

**2) Message Passing (mpsc) - The Safe Way to Share Data**

**Example: Waiters taking orders**

* The **kitchen** (receiver) waits for food orders.
* The **waiters** (senders) bring orders.
* Orders **don’t get mixed up** because each is **sent separately**.

**What Happens?**

* A thread **sends** food orders.
* The kitchen (receiver) **processes orders** one by one.
* No **data conflicts**, everything is delivered safely.

**Shared State with Arc<Mutex<T>> - Safe Shared Data**

**Example: A Bank Account**

* **Two ATMs** (threads) **withdraw money** from the same account.
* If both try to withdraw at the same time, we must **lock** the account to prevent mistakes.

**What Happens?**

* **Two threads** withdraw from the same account.
* Mutex<T> **locks** the account so only **one thread** can access it at a time.
* Arc<T> allows **multiple owners** of the account across threads.

**Threads** (std::thread) → Run multiple tasks at once.  
**Message Passing** (mpsc) → Safe way to share data **without conflicts**.  
**Shared State** (Arc<Mutex<T>>) → Share data safely **with locking**.

**Rayon: High-Performance Parallelism in Rust**

Rayon is a **data parallelism** library for Rust that makes it easy to convert **sequential iterations into parallel computations** while maintaining **safety and efficiency**.

**Why Use Rayon?**

1. **Automatic Thread Pooling** → Uses a **fixed-size thread pool** instead of creating threads per task, reducing overhead.
2. **Parallel Iterators (par\_iter())** → Process collections concurrently with minimal code changes.
3. **Work Stealing Algorithm** → Efficient task distribution across threads.
4. **Scalability** → Adapts to the number of CPU cores dynamically.
5. **Safe & Easy to Use** → No manual thread management needed.

**How Rayon Works**

Rayon provides **two core features**:

1. **Parallel Iterators (par\_iter())** → Automatic parallel processing.
2. **Parallel join() Execution** → Divide and conquer approach.