Interview Questions – Concurrency & Parallelism

🔹 Conceptual Questions

1. What is the difference between concurrency and parallelism?

- **Concurrency** is the ability of a system to handle multiple tasks at once (by managing context switching and task progress), even if not literally executing them at the same time.
- **Parallelism** means executing multiple tasks **simultaneously**, typically on multiple CPU cores.
- **Example:**
    - Concurrency: One-core CPU interleaving execution of multiple tasks.
    - Parallelism: Multi-core CPU executing multiple tasks in parallel.

---

2. How do threads differ from processes?

- **Processes** are isolated execution environments with their own memory space.
- **Threads** share the same memory space within a process but have their own stack.
- Threads are **lighter and faster** to create compared to processes.
- Communication between threads is easier (via shared memory), but this also introduces risks like race conditions.

---

3. What is a thread pool, and why is it preferred over creating new threads?

- A **thread pool** is a collection of reusable threads that are used to execute tasks.
- Benefits:
    - Reduces overhead of creating/destroying threads.
    - Prevents excessive thread creation (which can exhaust system resources).

    - Improves response time and system throughput.

- Common in web servers, background workers, async processing engines.

🔹 Practical Scenarios

4. How would you design a web server to handle thousands of concurrent requests?

- Use **asynchronous non-blocking I/O** (e.g., async/await in .NET or event loops in Node.js).
- Employ a **thread pool** for CPU-bound work.
- Use **reverse proxies/load balancers** to distribute load.
- Utilize **connection pooling**, **caching**, and **queue-based job processing** for scalability.

---

5. Describe how you would implement background job processing in a scalable system.

- Use a **message queue** (e.g., RabbitMQ, Kafka, Azure Queue).
- Workers consume tasks from the queue and process them asynchronously.
- Scale horizontally by increasing worker instances.
- Ensure **idempotency** and **retry logic** for failure handling.
- Monitor via dashboards and alerting tools.

---

6. How would you debug and resolve a deadlock in a multithreaded application?

- Use tools like **process dump analyzers**, **logging thread stacks**, or **Visual Studio Debugger**.
- Identify locks and resource acquisition order.
- Look for **circular wait conditions**.

- Fix by:
    - Always acquiring locks in a consistent global order.
    - Using **timeout-based locks**.
    - Minimizing lock scope (lock only what's necessary).
    - Switching to **concurrent collections** or lock-free data structures.

🔹 Pitfall Awareness

7. What is a race condition, and how can you prevent it?

- A **race condition** occurs when multiple threads access and modify shared data concurrently without proper synchronization, leading to unpredictable results.
- Prevention:
    - Use **locks/mutexes** (e.g., `lock` in C#).
    - Use **thread-safe collections**.
    - Leverage **atomic operations** (e.g., `Interlocked`).
    - Consider using **functional/stateless programming** where possible.

---

8. How do you ensure thread-safe operations in a shared-memory environment?

- Synchronize access to shared resources using:
    - `lock` or `Monitor` in C#
    - `ReaderWriterLockSlim` for read-heavy scenarios
- Use thread-safe APIs (e.g., `ConcurrentDictionary`)
- Prefer **immutable objects** and avoid shared state where possible.
- Leverage **Task Parallel Library (TPL)** and **async/await** to avoid manual thread management.

🔹 Bonus (Advanced)

9. How does the event loop work in Node.js or similar environments?

- Node.js uses a **single-threaded event loop** that:
    - Executes tasks from the **call stack**.
    - Offloads blocking tasks (like I/O or timers) to the **libuv thread pool**.
    - Callback functions are queued in the **event queue** and picked up when the stack is empty.
- Enables **high concurrency** without multi-threading.

---

10. What's the difference between parallelism using threads vs. async I/O?

- **Threads** are used for **CPU-bound** operations. Parallelism here means executing tasks across multiple CPU cores.
- **Async I/O** is used for **I/O-bound** operations. It doesn't block threads and uses callbacks/promises/futures to resume execution when I/O completes.
- Combining both:
    - Async I/O for fast, scalable networking
    - Threads (or parallelism) for compute-heavy operations