**LOUIE JAY A. CENIZA MARCH 25, 2024**

**CONCURRENCY (CHAPTER 13)**

**Why Concurrency?.** Concurrency is a technique used in software development to handle multiple tasks simultaneously. It allows programs to execute different parts of code concurrently, enabling better utilization of resources and improving performance. Concurrency decouples what needs to be done from when it gets done, which can enhance the throughput and structure of an application.

**Concurrency Defense Principles.** There are four types of principle and technique which are:

**Single Responsibility Principle:** which Concurrency-related code should be separated from other code because it has its own unique challenges and lifecycle. This separation helps manage the complexity of concurrent design and makes it easier to maintain and tune.

**Limit the Scope of Data:** To prevent interference between threads accessing shared data, restrict the access to shared data and protect critical sections using synchronization. Limiting the number of critical sections helps avoid errors and duplication of effort.

**Use Copies of Data:** Avoid sharing data between threads whenever possible. Instead, use copies of data or collect results independently in each thread and merge them later. This reduces the risk of synchronization issues and can improve performance by avoiding the need for synchronization.

**Threads Should Be as Independent as Possible:** Write threaded code in a way that each thread operates independently, with minimal or no shared data. Each thread should process its own data without relying on shared resources, reducing the need for synchronization. Partitioning data into independent subsets allows threads to operate independently, potentially improving scalability and performance.

By adhering to these principles and technique, developers can mitigate the challenges of concurrent programming and build more robust and scalable systems.

**Know Your Library.** When working with concurrency across different programming languages and libraries, it's crucial to leverage thread-safe data structures and libraries provided by the respective platforms. These libraries offer optimized implementations of common data structures and utilities for managing concurrent tasks and operations. Additionally, exploring executor frameworks or task queues can simplify the management of thread pools and task execution. It's essential to consider nonblocking or lock-free algorithms to improve scalability and performance in concurrent environments while minimizing shared mutable state to prevent concurrency bugs.

**Know Your Execution Models.** Understanding various execution models in concurrent programming is essential for developing robust and efficient systems. These models encompass concepts such as mutual exclusion, starvation, deadlock, live lock, and solutions like the Producer-Consumer, Readers-Writers, and Dining Philosophers problems. Mastering these fundamental algorithms equips developers with the skills to tackle complex concurrent challenges effectively.

**Testing Threaded Code.** Testing threaded code presents unique challenges due to the complexities introduced by concurrent execution and shared data. While it's impractical to prove code correctness, thorough testing can mitigate risks. Writing tests that cover various scenarios and running them frequently with different configurations and loads is crucial. Spurious failures should be treated as potential threading issues, and nonthreaded code should be validated first before tackling threaded implementations. Making threaded code pluggable and tunable facilitates testing under different configurations and performance scenarios. Additionally, running tests on different platforms helps uncover platform-specific issues.