Concurrency

Processes vs. Threads
✔ Processes:
Like runners in separate lanes → independent but must manage shared resources.
✔ Threads:
Like a team working in one lane → share resources within a process.

Synchronization Primitives
✔ Mutex (Mutual Exclusion):
Like a single special water bottle → only one thread can use it at a time.
✔ Semaphore:
Like a limited number of stretchers → multiple threads can access, but only up to a limit.

Critical Sections, Race Conditions, and Deadlocks
✔ Race Condition:
Two threads try to access the same resource at the same time, causing unexpected behavior.
Example: Two runners reach for the same water bottle at the same time → water spills.
✔ Deadlock:
Two threads wait for each other indefinitely, preventing progress.
Example:
Team Member A holds one stretcher but needs another.
Team Member C holds the other stretcher but also needs one more.
Neither can proceed → Deadlock!
📌 Proper synchronization (mutexes, semaphores) prevents race conditions and deadlocks in multi-threaded systems.

Why Use Concurrency?
✔ Improves speed & efficiency → Don't waste time waiting, do other tasks.
✔ Leverages modern multi-core CPUs → Keeps cores busy for better performance.
✔ Enhances UI responsiveness → Prevents freezing by running background tasks.

Why Is Concurrency Hard?
✔ Even experienced engineers struggle to avoid race conditions, deadlocks, and crashes.
✔ Requires careful synchronization (mutexes, semaphores) to prevent conflicts.

Real-World Failure: Therac-25 Radiation Overdose
✔ Issue: A race condition in the software caused incorrect radiation doses.
✔ Cause:
The system failed to synchronize shared variables.
High-dose mode accidentally activated, leading to severe overdoses.

🔴 Problem: Race Condition
✔ Threads t1 and t2 modify radiationLevel at the same time → No synchronization
✔ Since radiationLevel += 10 and radiationLevel -= 10 are not atomic operations, the final value may be unexpected or incorrect.
✔ Different runs might produce different results (unpredictable behavior).

✅ Solution: Mutex (std::mutex)
✔ std::lock_guard<std::mutex> lock(radiationMutex);

This locks the shared variable (radiationLevel) to prevent multiple threads from modifying it simultaneously.
✔ Ensures safe access to radiationLevel, preventing race conditions.
✔ Now, the final value of radiationLevel is always predictable and correct.

📌 Concurrency can be powerful but must be carefully managed to prevent catastrophic failures

Strategies for Handling Concurrency
1️⃣ Strategy 1 - Be Slow (Prioritize Correctness over Speed)
✔ Example: Regression Testing for ML Models
1. Copy models to a staging directory
   (a) Copy baseline models (10 min)
   (b) Copy test models (10 min)
2. Compare outputs using the same inputs.
3. Optimization Tip: (a) & (b) can run in parallel if copying is a bottleneck.

Key Idea: Prioritize accuracy and avoid concurrency issues by not rushing operations.

2️⃣ Strategy 2 - Isolate
✔ Keep shared resources separate to minimize conflicts.
✔ Instead of multiple threads sharing the same variable, give each thread its own copy.

3️⃣ Strategy 3 - Use Mutex (and Don't Forget to Unlock)
✔ Mutex (std::mutex) ensures only one thread modifies a resource at a time.
✔ Always unlock mutexes to avoid deadlocks.

```
std::mutex mtx;
void safeFunction() {
    std::lock_guard<std::mutex> lock(mtx); // Ensures mutex is unlocked automatically
    // Critical section
}
```

4️⃣ Strategy 4 - Use Tools
✔ Static Analysis Tools (e.g., ThreadSanitizer, Helgrind) detect concurrency bugs.
✔ Thread-safe libraries reduce manual handling of locks.

5️⃣ Strategy 5 - Put Comments
✔ Clearly document thread safety for each class/function.
✔ Examples of thread-safety comments: