1. Error Handling in Threads

Gracefully Handle Errors:

- Wrap thread code in a try-catch block to catch exceptions locally:

```
try {
    // Randomly throw error for demonstration
    if (taskId % 5 == 0) {
        throw std::runtime_error("Failed to process task " + std::to_string(taskId));
    }

    // Simulate processing time
    std::this_thread::sleep_for(std::chrono::milliseconds(100));

    // Successful processing log
    std::lock_guard<std::mutex> lock(logMutex);
    std::cout << "Thread " << threadId << " processed task " << taskId << std::endl;
} catch (const std::exception& e) {
    // Log error safely
    std::lock_guard<std::mutex> lock(logMutex);
    std::cerr << "Thread " << threadId << " error: " << e.what() << std::endl;
}
```

- Avoid throwing uncaught exceptions from threads as it may crash the program.

Safely Log Errors Without Race Conditions:

- Use std::mutex to guard shared logging resources:

```
std::lock_guard<std::mutex> lock(logMutex);
std::cout << "Thread " << threadId << " processed task " << taskId << std::endl;
```

2. Idle Threads / Load Balancing

Problem: Some threads may finish earlier, causing under-utilization.

Strategy – Work Queue (Dynamic Task Distribution):

- Use a **task queue** shared by all threads.
- Each thread picks tasks (chunks) from the queue until empty.
- Faster threads continue working instead of becoming idle.

Example:

```
std::lock_guard<std::mutex> lock(queueMutex);
if (taskQueue.empty()) return; // Exit when no tasks remain

taskId = taskQueue.front();
taskQueue.pop();
```

3. Threading Challenges & Solutions

Common Risks:

- **Race Conditions**: Multiple threads accessing shared data simultaneously.
- **Deadlocks**: Two or more threads waiting forever for each other's locks.
- **Resource Contention**: Too many threads causing CPU thrashing or memory issues.

Strategies to Avoid:

- Use **std::mutex** and **std::lock_guard** to protect shared data.
- Always **lock mutexes in the same order** to avoid deadlocks.
- Use **thread-safe containers** (e.g., std::atomic, std::queue with lock).
- Prefer **scoped locking (RAII)** over manual lock()/unlock().

4. Thread Pool

What is a Thread Pool?

- A thread pool is a set of pre-created worker threads that **wait for tasks** to execute.
- Tasks are placed in a **task queue**, and threads pick them up as they become available.

Benefits Over Fixed Thread Creation:

- **Avoids overhead** of creating and destroying threads repeatedly.
- **Efficient resource usage**: Threads are reused.
- **Scalable**: Adapts to varying workload; keeps CPU busy with minimal overhead.
- Useful in server applications, real-time systems, and repeated task processing.