Coding & Development / Debugging

Identify race conditions, deadlocks, or async timing issues in multi-threaded or asynchronous code.
Difficulty: Advanced
Model: GPT-4 / Claude / Gemini
Use Case: Concurrency Bugs, Race Conditions, Async/Await Issues
Updated: May 2026
Why This Prompt Exists
Most concurrency bugs are intermittent — they appear in production but not in development.

You get:

- race conditions that only happen under load
- deadlocks that freeze your application
- async timing issues that cause data corruption
- shared state modified by multiple threads/processes
- bugs that are nearly impossible to reproduce

But concurrency bugs are not random.

They follow predictable patterns.

- Race conditions: multiple operations reading/writing shared state
- Deadlocks: circular waiting for locks/resources
- Async timing: promises resolving in unexpected order
- Stale data: reading state after it’s been updated
- Lost updates: concurrent writes overwriting each other

Without analysis, concurrency bugs go undetected.

This framework forces AI to identify concurrency issues.

The Prompt

```
Assume the role of a concurrency debugging expert who finds race
conditions and deadlocks.

Your task is to analyze code for concurrency bugs.

Generate:

1. ISSUE TYPE IDENTIFICATION
   - Race condition
   - Deadlock potential
   - Async timing issue
   - Shared state mutation

2. PROBLEM LOCATION
   - Which lines/variables are involved

3. EXPLANATION
   - How the bug manifests
   - Conditions that trigger it

4. FIX RECOMMENDATION
   - Locking strategy (mutex, semaphore)
   - Atomic operations
   - Promise.all vs sequential
```

   - Immutable data patterns

5. FIXED CODE EXAMPLE

6. TESTING STRATEGY
   - How to reproduce and verify the fix

INPUTS:

Code (paste):
[PASTE CODE]

Concurrency Model:
[ASYNC/AWAIT / PROMISES / MULTI-THREADED / PROCESSES]

Observed Symptoms:
[E.G., "Data corruption occasionally," "App freezes," "Inconsistent results"]

Frequency:
[ALWAYS / OFTEN / RARELY / UNDER LOAD ONLY]

Environment:
[NODE.JS / BROWSER / PYTHON THREADING / OTHER]

RULES:
- Check for shared state modified by multiple async operations
- Check for missing await on promises
- Check for Promise.all vs. sequential when order matters

```
- Check for lock ordering (potential deadlocks)
- Check for race conditions in cached data
- Recommend specific synchronization primitives (not just "add a
lock")
- Consider using immutable data structures to avoid races
```

How To Use It

- Describe the symptoms even if you can’t reproduce them reliably.
- Check for shared state modified by multiple async operations — most common cause.
- Check for missing await on promises (fire-and-forget issues).
- Check if Promise.all is appropriate (order independence) vs. sequential (order matters).
- Consider using immutable data structures to eliminate race conditions entirely.

Example Input

Code:
let counter = 0;
async function increment() {
const current = counter;
await new Promise(resolve => setTimeout(resolve, 10));
counter = current + 1;
}
// Called multiple times concurrently

Concurrency Model: ASYNC/AWAIT (Node.js)

Observed Symptoms: Counter sometimes ends up lower than expected (e.g., 3 increments result in 2)

Frequency: OFTEN (about 30% of runs)

Environment: NODE.JS

Why It Works
Most concurrency bugs are subtle and intermittent.

This framework improves outcomes by forcing:

- issue type identification (precision)
- problem location (specificity)
- explanation (understanding)
- fix recommendation (action)
- testing strategy (verification)

Great concurrency debugging doesn't guess — it identifies patterns and prescribes fixes.

Build Better AI Systems

Subscribe for advanced prompt engineering, AI coding tools, debugging frameworks, and practical strategies for developers and engineers.

Carefully engineered prompts for people doing real work.

Share this:

- Share on Facebook (Opens in new window) Facebook
- Share on X (Opens in new window) X

See also The Error Message Explainer