---

**Category 1 — Foundations & Threading Primitives**

1. Implement a thread-safe bounded queue (fixed capacity) using mutex+condition_variable with FIFO fairness.
2. Build a reusable barrier for N threads that supports a post-barrier callback executed once per barrier trip.
3. Implement a reader-writer lock (shared/exclusive) with writer preference and test for starvation.
4. Write a thread-safe circular buffer supporting single-producer single-consumer (SPSC) and multi-producer multi-consumer (MPMC) modes (toggle via template).
5. Create a timed waitable event (manual-reset and auto-reset modes) supporting wait_for and wait_until.
6. Implement a parking-lot style spin + blocked waiter hybrid lock and measure latency vs std::mutex.
7. Build a reusable rendezvous primitive where M producers and N consumers must all arrive before proceeding.
8. Implement a thread-safe priority queue with blocking pop and cancellation support.
9. Build a counted latch (decrement-only) that can be reused after reset with minimal allocations.
10. Implement a concurrent reference-counted handle that safely supports upgrade from weak->strong across threads.

---

**Category 2 — Lock-Free Programming & Atomics**

11. Implement a lock-free singly-linked list supporting concurrent insert and remove (Harris-Michael algorithm) with hazard pointer reclamation.
12. Build a lock-free MPSC queue (following Vyukov or Lamport variants) and show ABA handling.
13. Implement a lock-free stack with Treiber algorithm and epoch-based reclamation.
14. Create a wait-free single-writer multi-reader register (value publish-subscribe) using atomics and versioning.
15. Implement a lock-free freelist allocator for fixed-size objects with safe memory reclamation.
16. Build a lock-free ring-buffer supporting multi-producer multi-consumer and explain memory order choices.
17. Implement a lock-free priority skiplist (simplified) supporting concurrent insert/delete.
18. Create an atomic double-word CAS wrapper (simulate if platform lacks) and explain use-cases.

19. Implement a lock-free reference-counted pointer (atomic RC) and discuss ABA and reclamation.
20. Build an atomic snapshot algorithm for multiple counters that yields a consistent snapshot.

---

**Category 3 — Thread Pools, Executors, and Scheduling**

21. Implement a high-performance thread pool supporting work-stealing between worker queues.
22. Build a priority-aware thread pool that schedules tasks by priority while avoiding starvation.
23. Implement a fork-join executor (parallel_for, parallel_reduce) with work-stealing and load balancing.
24. Create a cooperative multitasking scheduler using fibers (boost::context or userland stackless) on top of threads.
25. Implement a thread pool that supports task affinities (pin tasks to a specific worker or core).
26. Build a timed/scheduled executor supporting delayed tasks and recurring periodic tasks with drift correction.
27. Implement a bounded thread pool with rejection policies (block, drop, grow) and benchmarking harness.
28. Create a dependency-aware executor that accepts tasks with DAG dependencies and schedules respecting dependencies.
29. Implement a thread pool that can dynamically resize based on workload and CPU utilization heuristics.
30. Build a hierarchical executor that partitions tasks by latency sensitivity (foreground vs background).

---

**Category 4 — Synchronization Patterns & Advanced Locks**

31. Implement a combination of spinlock + mutex (adaptive lock) that switches strategy based on contention.
32. Build a read-copy-update (RCU) simplified framework with readers-side lockless reads and graceful updates.
33. Implement a two-phase commit-style synchronization primitive for coordinating changes across threads.
34. Create a distributed lock (local simulation) that supports exclusive and shared modes using TCP sockets.
35. Implement a priority inheritance mutex to avoid priority inversion in a simulated RT environment.

36. Build a transactional memory simulator (software TM) with conflict detection and rollback of small critical sections.
37. Implement a semaphore with async wait support (integration with futures/promises).
38. Create an optimistic concurrency control wrapper for an object: readers proceed lock-free with validation, writers take locks.
39. Implement an epoch-based reclamation system for safe memory reclamation in concurrent containers.
40. Build a scalable barrier using tree-based combining to reduce contention for large thread counts.

---

**Category 5 — Locking Strategies & Performance Engineering**

41. Design and implement a microbenchmark suite to compare different mutex implementations (std::mutex, spinlock, pthread rwlock).
42. Implement a cache-aware concurrent hash map with per-bucket locks and probe optimization for low false sharing.
43. Build a sharded concurrent LRU cache with per-shard locking and global size limit enforcement.
44. Implement a contention-adaptive skiplist where node-level locks escalate under contention.
45. Create a concurrent hash set using split-ordered lists for lock-free resizing.
46. Implement a low-latency alarm/timer wheel for scheduling many short-lived timers efficiently.
47. Build a memory allocator optimized for multithreaded allocations (per-thread caches + central depot).
48. Implement an intrusive concurrent deque optimized for producer-consumer patterns.
49. Create a throughput vs latency trade-off harness for a concurrent queue and analyze results.
50. Implement a thread-local object pool with cross-thread zombie reclamation.

---

**Category 6 — Async, Futures/Promises, and Composability**

51. Implement a composable futures library (a la std::future + continuations) that supports then(), catch(), and combine_all().
52. Build an awaitable/async framework (co_await/co_return style) on top of coroutines for task suspension/resume.
53. Implement a promise/future that supports cancellation and propagation of cancellation to ancestors.
54. Create a when_any / when_all combinator that preserves task cancellation and partial results.

55. Build a flow-control backpressure-enabled async pipeline between producer and consumer tasks.
56. Implement a lightweight reactive stream abstraction with operators map/filter/flat_map and subscription control.
57. Create an asynchronous channel with backpressure and multiple readers/writers that integrates with coroutines.
58. Implement a cooperative fiber scheduler with awaitable timers and I/O readiness integration.
59. Build a timeout wrapper for arbitrary futures that supports fallback continuation on timeout.
60. Implement a distributed future (simulate across processes) that can migrate or resume on another node.

---

**Category 7 — IO Concurrency, Networking & Multiplexing**

61. Implement an asynchronous reactor (epoll/kqueue/select abstraction) with thread pool for handler execution.
62. Build a non-blocking HTTP client supporting pipelined requests with connection pooling.
63. Implement a concurrent TCP server with per-connection state machine and graceful shutdown.
64. Create a high-throughput message broker (in-memory) with topics, subscriptions, and acknowledgements.
65. Implement zero-copy message passing between threads using mmap/shared memory and ring buffers.
66. Build an efficient scatter-gather IO system that batches small writes into larger syscalls.
67. Implement a prioritized networking queue that shapes outgoing traffic per-connection priority.
68. Create a reliable UDP protocol (simple rUDP) with ACKs, retransmit, congestion control simulation.
69. Implement an async file IO framework that batches fsyncs and reduces syscalls.
70. Build a backpressure-aware multi-protocol gateway that bridges producers (sockets) to consumers (worker threads).

---

**Category 8 — Concurrency & Memory Model, Ordering**

71. Prove correctness: implement and test Dekker's or Peterson's algorithm using C++ atomics with proper memory orders.
72. Implement and demonstrate a lock-free data structure requiring release-acquire fences and justify ordering choices.

73. Build a small litmus test suite generator that emits concurrent C++ code exploring reordering effects across memory orders.
74. Implement explicit memory fences (atomic_thread_fence) to coordinate a synchronization pattern and validate on x86/ARM (simulation).
75. Create a publish-subscribe memory model demo that shows consequences of relaxed ordering on visibility.
76. Implement a sequence lock (seqlock) with writers and readers and handle rollback conditions.
77. Build a concurrent counter using fetch_add, fetch_sub and implement scalable reduce for many producers.
78. Implement a lock-free double-checked initialization pattern correctly across platforms.
79. Create an atomic flag-based rendezvous with release/acquire semantics and test causal relationships.
80. Implement versioned object updates with atomic stamping and validate ABA avoidance strategies.

---

**Category 9 — Testing, Debugging & Tooling for Concurrency**

81. Build a stress-test harness that fuzzes thread schedules, injects delays, and runs thread sanitizer friendly scenarios for a concurrent container.
82. Implement a deterministic thread scheduler (record/replay) to reproduce concurrency bugs reliably.
83. Create a race detector that instruments reads/writes in user code (lightweight sampling-based).
84. Build a visualizer that records thread events (lock acquire/release, thread start/end) and outputs a trace timeline (JSON).
85. Implement automatic deadlock detection for a program by tracking lock graphs at runtime.
86. Create a chaos-testing framework that randomly kills worker threads and validates system resilience.
87. Implement a concurrency unit-test DSL to express invariant-based tests that run under many schedules.
88. Build an allocator profiler that reports cross-thread allocation hotspots and false-sharing candidates.
89. Implement a lightweight snapshot profiler that samples call stacks and thread states for contention hotspots.
90. Create a “heisenbug” reduction tool that bisects code to find minimal inputs that trigger concurrency issues.

---

**Category 10 — Distributed Concurrency, Consensus & Fault Tolerance**

91. Implement a simplified Paxos proposer/acceptor simulation for learning (single-decree).
92. Build a leader election algorithm (Bully or Raft leader election subset) across simulated nodes with thread-per-node.
93. Implement a distributed lock manager using a quorum-based scheme with retries and backoff.
94. Create a replicated state machine skeleton that applies commands in total order (basic Raft log apply).
95. Implement eventual consistency for a key-value store with vector clocks and anti-entropy (gossip) synchronization.
96. Build a partition-tolerant coordinator that handles network partitions and heals state when connectivity returns.
97. Implement a lease-based leader with clock skew handling and safe leader takeover.
98. Create a fault-injecting harness (message drop, reorder, delay) and demonstrate recovery of your distributed algorithm.
99. Implement distributed snapshots (Chandy-Lamport) across simulated threads/nodes with channel recording.
100. Build a simple distributed transaction commit protocol (two-phase commit) with participant crash/recovery simulation.