THE ART OF CONCURRENT PROGRAMMING

Locks, Atomics, Lock-Free Systems and High-Performance Concurrent Software

ROBERT HARTLEY

The Art of Concurrent Programming

Locks, Atomics, Lock-Free Systems and High-Performance Concurrent Software

Steve Publications

This book is available at
https://leanpub.com/theartofconcurrentprogramming

This version was published on 2026-09-15

Contents

Locks, Atomics, Lock-Free Systems and High-Performance Concurrent Software

This book is a complete technical guide to building correct, efficient concurrent software for production systems. It covers the full stack from hardware architecture and memory models through synchronization primitives, lock-free algorithms, debugging techniques, performance engineering, and real-world architecture patterns. The audience is experienced software engineers who need to design, implement, benchmark, and operate reliable concurrent systems at scale, and who want to understand not just what works, but why, what can go wrong, and how to choose between alternatives when trade-offs matter.

Introduction

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

What This Book Is About

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

A Word About Trade-offs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

How to Read This Book

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Why This Depth?

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Chapter 1: Concurrency vs Parallelism: Defining the Problem

What Concurrency Actually Means

Concurrency and parallelism are not synonyms, and confusing them leads to confused thinking about what your code actually does. The distinction is simple once stated, but its implications run deep.

Parallelism is about doing multiple things at the same time. If you have eight CPU cores and you run eight independent computations simultaneously, one on each core, that is parallelism. Parallelism is a property of hardware execution: multiple operations actually advancing together in wall-clock time.

Concurrency is about dealing with multiple things at once. It is a structural property of your program. A program is concurrent if it manages multiple independent flows of execution whose relative timing is not fixed by the program itself. Those flows may run in parallel on multiple cores, or they may run one at a time on a single core through interleaving, or some combination of both. The key point is that the program must be correct regardless of how those flows interleave, within the guarantees of its execution environment.

The classic illustration is a single-threaded event loop. A web server that accepts connections, reads requests, processes them, and writes responses using a non-blocking I/O event loop is concurrent. Multiple client connections are being handled simultaneously in the sense that the program is dealing with multiple independent flows of execution. But at any instant, only one is actually executing. The event loop switches between them, responding to readiness notifications. The program must be correct even though it does not control when or how these flows interleave, a response must not contain data from another request, a buffered write must not overwrite another connection's data, and so on. This is concurrency without parallelism.

Modern servers typically use both. They run a pool of worker threads, each executing event loops or handling requests, and they run on multiple cores so that threads actually execute in parallel. But the program must be correct

at both levels: correct under interleaving within each thread's event loop (concurrency), and correct when threads execute simultaneously on different cores (parallelism).

Understanding this distinction matters because it tells you what guarantees you can rely on. In a purely concurrent context without parallelism, like a single-threaded event loop, you never have to worry about data races in the sense of two threads writing to the same memory simultaneously. You still have concurrency problems: state management, interleaving of operations from different logical flows, and reasoning about timing. But you do not need locks, atomics, or memory barriers because there is only one thread of execution. When you add parallelism, you bring the full complexity of shared-memory concurrency.

Concurrency Without Parallelism

The notion that concurrency can exist without parallelism is worth emphasizing because it clarifies what concurrency is fundamentally about: managing complexity arising from multiple interleaved flows of execution.

Consider a Unix shell running a pipeline: `grep pattern file | sort | uniq -c`. The three processes communicate via pipes, each with its own execution timeline. The shell does not control whether `sort` finishes before `uniq` starts reading, or whether `grep` produces all its output before `sort` begins. The system is concurrent: multiple independent flows of execution interacting through shared buffers (the pipes). Whether those flows actually execute simultaneously depends on the scheduler and available cores.

Event-driven frameworks extend this idea into user space. Node.js runs JavaScript code in a single thread. When it encounters an I/O operation, it registers a callback with the operating system and continues executing other code. When the I/O completes, the callback is scheduled to run. Multiple operations are in flight concurrently, and the program must maintain invariants despite the non-deterministic timing of their completion. There is no parallelism in the JavaScript execution itself (barring Web Workers), but there is significant concurrency.

Go's goroutines provide another perspective. Goroutines are lightweight, user-space threads managed by the Go runtime scheduler. By default, the runtime schedules goroutines across a limited set of operating system threads

(controlled by `GOMAXPROCS`). If `GOMAXPROCS` is one, all goroutines execute sequentially on a single OS thread, preemptively scheduled by the Go runtime. Yet Go programs with goroutines are fully concurrent. They must use the same synchronization primitives, channels, mutexes, atomics, regardless of whether `GOMAXPROCS` is one or thirty-two, because the Go memory model makes the same guarantees in both cases, and the correctness of the program should not depend on deployment configuration.

The broader lesson is that concurrency is fundamentally a design problem. It arises whenever your program needs to handle multiple things that do not have a fixed timing relationship. Parallelism is an implementation detail, a way to get more work done, but concurrency exists even when you have no parallel hardware at all.

The Cost of Getting It Wrong

What happens when concurrent code is wrong? The answer depends on the nature of the bug, but the common thread is that failures are harder to diagnose than sequential bugs by orders of magnitude.

A sequential bug has a deterministic trigger. If you pass null to a function that expects a non-null pointer, it crashes every time, at the same location, with the same stack trace. You can reproduce it, you can minimize it to a failing test case, and you can fix it. The relationship between the bug and its manifestation is direct and stable.

A concurrency bug is different. A data race between two threads accessing shared memory without synchronization may cause a crash only when the threads interleave in a particular way. That interleaving may happen once every thousand runs, or once every million. It may never happen on your developer machine with its four cores, but it happens constantly in production with sixty-four cores and higher load. It may stop happening when you add logging, because the logging changes timing enough to prevent the problematic interleaving. This is the “Heisenbug” phenomenon: observing the system changes its behavior.

Deadlocks are another category. Two threads each holding one lock and waiting for the other may deadlock only when they execute a specific interleaving of their operations. This interleaving might require that Thread A acquires Lock X before Thread B starts, that Thread B acquires Lock Y before

Thread A tries to acquire it, and that the timing is such that neither thread releases its lock before the other blocks. In low-load testing, threads may run sequentially enough that this never happens. Under load, with many threads competing for locks, the problematic interleaving becomes probable.

Performance failures are perhaps the most insidious. Consider false sharing: two threads modify different variables that happen to be on the same cache line. Every time one thread writes, the other thread's cache line is invalidated, forcing a reload. The program is logically correct, no data race exists because each thread accesses its own variable. But performance degrades dramatically as the number of cores increases, sometimes by a factor of ten or more, because cache-coherence traffic serializes what should be independent updates. There is no crash, no wrong result, just "it's slow and we don't know why."

These bugs are expensive because they consume time. A production incident caused by a race condition might take hours or days to diagnose. Developers who are not comfortable with concurrency tend to avoid concurrent designs, leading to sequential bottlenecks that limit scalability. Teams ship mutex-based solutions that work under testing conditions but lock convoy under production load, degrading latency for all users.

The cost is not just technical. It is organizational. Teams that have been burned by concurrency bugs become reluctant to add concurrency, even when their systems genuinely need it. Systems end up running single-threaded event loops that cannot take advantage of available cores, or they use connection-per-thread models that hit thread limits under load. The fear of concurrency bugs leads to architectural constraints that limit what the system can achieve.

Why Modern Hardware Demands Concurrency

There is a pragmatic reason concurrency has become unavoidable: single-threaded performance stopped scaling.

For roughly twenty years from the mid-1970s to the mid-2000s, processor performance improved rapidly due to increasing clock speeds and architectural improvements (pipelining, out-of-order execution, branch prediction). A single-threaded program running on a processor in 2000 could expect to run several times faster on a processor from 2005. This trend is sometimes called "free lunch": you did not need to change your code to get faster execution.

Then it stopped. Clock speeds hit a wall, physical limits on heat dissipation and power consumption, around 3-4 GHz for mainstream processors. The industry's response was to put more cores on each chip. A 2005-era desktop had one core. A 2025-era workstation might have sixty-four cores on a single socket, and servers might have two or more such sockets.

The implication is direct: if you want your application to utilize modern hardware, it must be concurrent. A single-threaded program uses one core and leaves the rest idle. To handle more load, you must either run more instances of your application on different cores or redesign it to use multiple threads or processes. There is no going back to the free lunch.

This hardware trend has compounded with other factors:

Network latency has not decreased proportionally with CPU speed improvements. In the 1980s, a round-trip to a local network server might take roughly the same number of CPU cycles as today. But because CPUs are orders of magnitude faster now, each millisecond of latency represents millions of wasted cycles. A single-threaded program that blocks on a network call for 10 milliseconds wastes enough time to execute tens of millions of instructions. Concurrent programs keep other work going while waiting for I/O.

Memory latency tells a similar story. Accessing main memory takes roughly 100 nanoseconds on modern systems, which is about 200-300 CPU cycles on a 2-3 GHz processor. If a program stalls waiting for memory, a concurrent program can switch to another thread that has its data in cache. Modern CPUs do this speculatively within a core using out-of-order execution, but across multiple threads, explicit concurrency provides the same benefit at a higher level.

Workload characteristics have changed. Modern applications handle many clients simultaneously, each with different timing profiles. A web server receiving thousands of concurrent connections cannot efficiently handle them all sequentially; some will be waiting for database responses, some for external APIs, some for disk I/O. Concurrent handling allows the server to make progress on all of them simultaneously, improving throughput and tail latency.

The hardware argument is not subtle: concurrency is no longer optional for performance-sensitive software. The question is not whether to use concurrency, but how to use it correctly and efficiently.

The Trade-off Landscape

Before diving into the technical details, it is useful to frame the space of trade-offs that you will encounter throughout this book. Concurrency is not a collection of independent techniques; it is a landscape of trade-offs, and understanding where you stand in that landscape determines what techniques are appropriate for your situation.

Correctness versus complexity. Simple synchronization, a single global mutex protecting all shared state, is easy to get correct. It eliminates data races by construction. But it introduces a global bottleneck: only one thread can proceed at a time, regardless of how many cores you have. More sophisticated designs using fine-grained locking, lock-free algorithms, or sharding can provide better scalability, but they add complexity and opportunities for bugs. The fundamental question is: what level of correctness risk can you afford to take in exchange for performance?

Latency versus throughput. A system optimized for throughput might batch operations, process them in bulk, and achieve high aggregate performance, but individual operations may experience high latency as they wait in queues. A system optimized for latency might prioritize getting each individual operation done quickly, even if it reduces overall throughput due to context switching or underutilized resources. Concurrency decisions affect this trade-off directly. Lock-free data structures often provide better tail latency than lock-based ones because a slow thread does not block all others. But they may have higher average latency under low contention due to cache-coherence overhead.

Scalability versus predictability. Some concurrent designs scale well as you add cores or increase load, but their behavior becomes harder to predict and debug. Lock-free algorithms can continue making progress as cores are added, but they exhibit complex interactions under contention that are difficult to characterize. Lock-based designs with clear critical sections are easier to analyze and reason about, even if they eventually bottleneck. Real-time and safety-critical systems often prefer predictability over scalability.

Simplicity of interface versus power. High-level concurrency abstractions, actor models, channel-based communication, async/await patterns, can make concurrent code easier to write by hiding low-level details. But they may not map efficiently to the underlying hardware, or they may impose constraints

that make certain performance optimizations difficult. Low-level primitives like atomics and spinlocks give you fine-grained control, but they require deep understanding of memory models and hardware behavior to use correctly.

Portability versus optimization. Algorithms that work correctly on one platform may fail on another if they rely on implicit memory-ordering guarantees that are not part of the language specification. Code that takes advantage of x86's strong memory model may break on ARM. Code that assumes a garbage collector may not work in environments where you need deterministic memory management. Portability requires either accepting the weakest common denominator or encoding platform-specific behavior carefully.

Throughout this book, each technique I discuss will be evaluated against these trade-offs. The goal is not to declare any one approach universally superior. The goal is to give you enough understanding to make informed decisions based on your actual constraints, workload characteristics, and operational requirements.

The rest of this book explores these trade-offs in depth. Part I establishes the hardware foundation that makes all these trade-offs necessary. Understanding how CPUs, caches, and memory systems actually work is not an academic exercise, it explains why naive concurrent code fails, why optimizations that make sense in theory perform poorly in practice, and why certain patterns recur across different domains and languages.

Chapter 2: Processes, Threads, and the Operating System

The Unix Process Model

The Unix process model is the foundation on which all concurrent execution in Unix-like operating systems is built. To understand concurrency, you need to understand what the operating system considers an independent unit of execution, what resources it isolates, and what it shares.

A process is an executing instance of a program with its own address space, file descriptors, and execution state. The address space, the range of virtual memory addresses the process can use, is the most important resource a process owns. It contains the program's code, global variables, heap, stack, and memory-mapped files. The operating system uses virtual memory to isolate processes from each other: each process sees its own private address space, and memory corruption in one process does not directly affect another.

When a process calls `fork()`, the kernel creates a child process that is a copy of the parent. The child gets a duplicate address space with identical contents, all the variables, heap allocations, and code are copied. In practice, Linux uses copy-on-write (COW): the parent and child initially share physical pages, and the kernel copies a page only when one of the processes writes to it. This makes `fork()` fast even for processes with large address spaces, because the expensive copy happens only if actually needed. The child typically calls `exec()` to replace its address space with a new program, in which case no copying occurs at all.

Processes are heavyweight. Creating a process requires allocating a new address space, setting up page tables, duplicating file descriptor tables, initializing kernel data structures, and performing other setup work. On a modern Linux system with a modest heap, `fork()` takes roughly thirty-five microseconds [1]. That is fast in absolute terms, but it is orders of magnitude slower than calling a function. If your application needs to create many

execution contexts, such as a web server handling thousands of connections, creating one process per connection quickly becomes impractical.

Processes also consume significant memory. Even with COW, each process requires kernel data structures and page tables. The Linux kernel stores per-process state in a `task_struct`, which is several kilobytes. Page tables consume memory proportional to the size of the address space. On 64-bit systems with large virtual address ranges, page tables can consume substantial memory even when most of the address space is empty.

The advantage of this isolation is clear: if one process crashes due to a memory corruption bug, other processes are unaffected. A segfault in a worker process does not bring down the entire application. For systems that handle untrusted code or need strong fault containment, processes are the right abstraction.

Kernel Threads and User Threads

Threads were introduced to address the overhead of processes while preserving the benefits of concurrent execution. A thread is a unit of execution within a process that shares the process's address space, file descriptors, and other resources with other threads in the same process. Threads have their own stack, program counter, and register state, but they can directly read and write each other's variables through the shared address space.

The distinction between "kernel threads" and "user threads" is important because it determines who manages scheduling: the operating system kernel or the application runtime.

Kernel threads are managed directly by the operating system. Each kernel thread is an independent scheduling entity that the kernel's scheduler can place on any available CPU core. When a kernel thread blocks on I/O, the kernel can schedule another thread on that core without any involvement from the application. Kernel threads are what you get when you call `pthread_create()` on Linux or create a `std::thread` in C++.

On Linux specifically, there is no distinction between processes and threads at the kernel level. Both are implemented using the same `task_struct` and created through the same `clone()` system call. The difference is only in the flags passed to `clone()`. `fork()` calls `clone()` with flags that create a new address space and new process identity. `pthread_create()` calls `clone()`

with `CLONE_VM` and `CLONE_THREAD` flags, which make the new task share the caller's address space and thread group. From the kernel's perspective, a thread is simply a task that shares resources with other tasks [2].

This design has consequences. Because Linux threads are full kernel tasks, they are scheduled by the kernel just like processes. They benefit from the kernel's sophisticated scheduling algorithms, NUMA awareness and CPU affinity management. But they also carry the cost of kernel-level scheduling: context switches between threads involve transitioning into kernel mode, saving and restoring registers, updating scheduler data structures, and potentially flushing parts of the CPU cache or TLB.

User threads are managed by a runtime library or application framework without direct kernel involvement. The runtime multiplexes many user threads onto a smaller set of kernel threads. When a user thread blocks, the runtime switches to another user thread on the same kernel thread without involving the kernel. This can be much cheaper than a kernel-level context switch because it stays entirely in user space.

The classic example of user threads is Java's green threads, which were used in early Java implementations before Java 1.2. The JVM managed all thread scheduling internally and mapped Java threads to a single OS thread. This provided cheap thread creation and context switching, but it had a fundamental limitation: if one green thread performed a blocking system call, the entire process blocked because the single underlying OS thread was blocked.

The M:N threading model, mapping M user threads onto N kernel threads, attempts to combine the benefits of both approaches. Green threads provided M:1. Modern approaches like Go's goroutines and Erlang's lightweight processes use M:N with work-stealing schedulers that keep all available CPU cores busy while still providing cheap user-level scheduling.

The trade-offs are clear:

Kernel threads provide true parallelism because the kernel can schedule different threads on different cores simultaneously. They integrate with the operating system's scheduling, priority management, and resource accounting. But they are relatively expensive to create and switch between, and the system has a practical limit on how many can exist. On 64-bit Linux, you can create roughly thirty-two thousand threads per process before running into limits [3], but long before that, context-switching overhead and memory consumption degrade performance.

User threads are cheap, creating and switching between them can cost orders of magnitude less than kernel threads. Go's runtime can create millions of goroutines and switch between them in approximately 170 nanoseconds, compared to 1-2 microseconds for Linux kernel thread context switches [3]. But they require sophisticated runtime support to handle blocking operations and to map efficiently onto available cores.

For most production systems today, kernel threads (or M:N models that expose kernel threads to the system) are the dominant approach. Pure user-thread models have fallen out of favor because they struggle to utilize multi-core hardware efficiently. But understanding both models is essential because modern runtimes like Go and the JVM blend elements of both.

Context Switching and Its True Cost

Every time the operating system switches execution from one thread to another, it performs a context switch. Understanding the cost of a context switch is essential for designing concurrent systems because context switching is a non-productive operation: the CPU is not doing useful work during a context switch, it is managing state.

A context switch involves several steps. The kernel saves the current thread's register state (general-purpose registers, program counter, stack pointer, floating-point registers, etc.) to the thread's kernel stack. It updates scheduler data structures, potentially changing the thread's run queue state. It may switch the memory context if switching between processes with different address spaces, which involves updating the page table base register and potentially flushing parts of the TLB (Translation Lookaside Buffer). It restores the next thread's register state from its saved context. Finally, it returns from the kernel to user mode, resuming execution of the next thread.

The total cost depends on several factors:

For thread-to-thread context switches within the same process on Linux, the cost is roughly 1.2-1.5 microseconds when both threads are pinned to the same core and their data is still in cache [3]. This includes saving and restoring registers, updating scheduler structures, and returning to user mode. When threads can migrate between cores, the cost increases to roughly 2.2 microseconds or more because the scheduler must account for cache effects and potentially flush per-core state [3].

For process-to-process context switches, the cost is higher because the kernel may need to switch page tables and flush TLB entries. The exact cost depends on whether hardware TLB shootdown is needed, whether the processes share page table entries through COW, and whether the CPU supports PCID (Process Context Identifiers), which can avoid TLB flushes in some cases.

These numbers, 1-5 microseconds, may seem small. They are, in absolute terms. But consider what happens when your application is context-switching heavily. If a CPU core is spending 10 percent of its time in context switches, that is 10 percent of capacity doing no useful work. If you have high lock contention and many threads spinning or blocking, context-switching overhead can consume a significant fraction of your CPU budget.

More importantly, context switches have indirect costs that are often larger than the direct overhead. When the kernel switches to a different thread, that thread's data may no longer be in the CPU cache or TLB. The new thread must reload its data from memory, which takes hundreds of CPU cycles per cache miss. If the previous thread was working with hot data and the new thread is working with cold data, the cache pollution can degrade performance for both. These cache effects can easily multiply the effective cost of a context switch by five to ten times.

There is also the scheduler latency: the time between when a thread becomes runnable and when the scheduler actually runs it. Linux's Completely Fair Scheduler (CFS) uses a target latency of roughly 4-6 milliseconds [4]. If a thread blocks and becomes runnable again, it may not run immediately; it waits for its fair share. In systems with strict latency requirements, real-time systems, low-latency trading, gaming servers, this scheduling latency can be problematic.

For these reasons, high-performance concurrent systems try to minimize unnecessary context switches. Techniques include:

- Using thread pools so that threads are long-lived and do not need to be created and destroyed for each task
- Using event-driven I/O (epoll, kqueue, io_uring) so that a thread can handle many connections without blocking and triggering a context switch
- Pinning threads to specific cores using CPU affinity to reduce cache invalidation when threads are rescheduled
- Using busy-waiting (spinning) for very short wait times when the expected wait is shorter than the cost of a context switch

None of these techniques is free. Busy-waiting wastes CPU cycles. Thread pools complicate code. CPU affinity can lead to load imbalance if work distribution is uneven. But understanding the cost of context switching helps you evaluate these trade-offs.

Scheduling Policies and Predictability

The operating system's scheduler determines which threads run on which cores and for how long. The scheduler's behavior affects your application's performance, latency characteristics, and even correctness in some cases.

Linux's default scheduler for normal tasks is the Completely Fair Scheduler (CFS), introduced in kernel 2.6.23. CFS does not use fixed time slices or priority-based run queues. Instead, it maintains a red-black tree of runnable tasks sorted by "virtual runtime", a measure of how much CPU time each task has received, weighted by its nice value (priority). The scheduler always runs the task with the lowest virtual runtime, which ensures that over time, each task receives CPU time proportional to its weight [4].

CFS is designed for fairness and interactive responsiveness, not for predictable latency. It targets a scheduling latency of roughly 4-6 milliseconds, meaning that if all runnable tasks are equally entitled to CPU time, each one will get a turn within that window. This is good for desktop workloads and general-purpose servers, where fairness and throughput matter more than strict latency bounds. But it means that in a heavily loaded system with many runnable threads, your latency-sensitive thread might wait several milliseconds before running, and you cannot control this deterministically.

Linux also provides real-time scheduling policies: `SCHED_FIFO` and `SCHED_RR`. These are priority-based preemptive schedulers. A `SCHED_FIFO` task runs until it blocks, yields, or is preempted by a higher-priority task. A `SCHED_RR` task runs for a fixed time quantum and then yields to the next task at the same priority. These policies provide more predictable latency for high-priority tasks, but they come with risks. A high-priority real-time task that does not block can starve all lower-priority tasks, including the kernel's background work. If a real-time task holds a lock that a lower-priority task needs, and a medium-priority task preempts the lower-priority task, you get priority inversion, a topic we will explore in Chapter 15.

The scheduling policy you choose depends on your requirements:

- For most server workloads, the default CFS scheduler is fine. Your threads should do useful work between synchronization points, and the scheduler will distribute CPU time fairly.
- For latency-sensitive operations, you might use CPU affinity to pin latency-critical threads to specific cores, or use `SCHED_FIFO` carefully for threads that must run within strict latency bounds.
- For real-time systems that must guarantee worst-case latency, you need a real-time operating system (RTOS) or a carefully configured Linux system with preemption patches (PREEMPT_RT), lock-free data structures, and strict priority management.

An important detail: scheduling is non-deterministic. Even with the same code, the same inputs, and the same hardware, your threads may execute in different orders on different runs because the scheduler makes decisions based on timing, load, and other factors that vary between runs. This non-determinism is the root cause of many concurrency bugs: a race condition may never manifest during development because the scheduler happens to interleave threads in a benign order, but it crashes in production when the interleaving is different.

Good concurrent code does not rely on the scheduler behaving in any particular way. It is correct for all valid interleavings of thread execution, within the guarantees of the memory model. Testing for all interleavings is impossible, but techniques like stress testing, race detection, and model checking help expose races that depend on unusual scheduling behavior.

M:N Thread Models and Goroutines

Go's goroutine scheduler is the most prominent modern example of an M:N threading model, mapping many lightweight user threads (goroutines) onto a smaller set of OS threads. Understanding how it works provides insight into how concurrency can be made practical at large scale.

The Go runtime uses a G-P-M model:

- G (goroutines): Lightweight execution units. Each goroutine starts with a small stack (roughly 2 KB) that grows and shrinks dynamically. Millions of goroutines can exist in a single process.

- P (processors): Logical processors that represent permission to execute. The number of Ps is limited by `GOMAXPROCS`, which defaults to the number of available CPU cores. Each P maintains a local run queue of goroutines.
- M (machine/OS threads): Actual operating system threads that execute goroutines. Each M is bound to a P while executing goroutines.

When a goroutine is created, it is placed in the local run queue of the current P. The P's associated M picks goroutines from its local queue and executes them. When a goroutine blocks (on I/O, a channel operation, or a mutex), the runtime parks it and the M moves to execute another goroutine from the queue. If the blocking operation is a system call, the M may detach from the P, allowing another M to attach and continue executing goroutines from that P's queue [5].

Work stealing is the key mechanism that keeps cores busy. When a P's local queue is empty, its M does not immediately block. Instead, it attempts to steal goroutines from other Ps. Specifically, it picks a random P and steals half of the goroutines from that P's local queue. This decentralized load balancing avoids the need for a global lock on a single queue, which would become a bottleneck under high concurrency. The runtime also maintains a global run queue as a fallback; when goroutines are created and the local queue is full, they are moved to the global queue, from which idle Ps can pull work [5].

The scheduler also uses spinning threads to reduce latency. When an M blocks waiting for work, instead of immediately going to sleep (which would require a context switch and scheduler wakeup), it spins for a short time trying to find work. This reduces the latency of starting newly created goroutines because another M might pick them up immediately without waiting for the kernel scheduler. The number of spinning threads is bounded to prevent wasting CPU cycles [5].

Preemption is handled cooperatively and via async signals. Go uses function call trampolines to check whether a goroutine should be preempted, and for long-running goroutines without function calls, it can use signals to inject preemption checks. This ensures that no single goroutine can monopolize a core.

The performance characteristics of this model are impressive. Goroutine context switches are roughly 170 nanoseconds [3], about an order of magnitude cheaper than kernel thread context switches. Goroutine creation is similarly cheap, you can create a million goroutines in a fraction of a second. Memory

overhead per goroutine is low because stacks start small and grow only as needed.

But the M:N model has trade-offs:

- System calls are expensive. When a goroutine executes a system call, it blocks the underlying M. If many goroutines are making system calls, you need many Ms, which means you are back to having many kernel threads with their overhead. Go mitigates this with netpolling, handling network I/O without blocking system calls, but other I/O operations still require kernel threads.
- Lock contention on runtime data structures. The scheduler itself uses locks and atomic operations to manage goroutine queues, and under extreme concurrency, these can become bottlenecks.
- Debugging complexity. Stack traces from goroutines are managed by the runtime, not the kernel, which can complicate debugging with tools like GDB.
- Limited integration with external threading libraries. If your Go code calls C libraries that use threads, those threads are not managed by the Go scheduler and can cause unexpected blocking.

Despite these trade-offs, the M:N model has been enormously successful. It has made high-level concurrency practical in Go: developers can spawn goroutines freely without worrying about thread pool sizes or connection-per-thread scaling limits. Other languages have adopted similar approaches: Java's Project Loom aims to add virtual threads with comparable characteristics, and Rust has several M:N runtime libraries.

The broader lesson is that abstraction over the kernel threading model can make concurrency more practical without sacrificing performance, as long as the runtime handles blocking operations and load balancing efficiently. For systems where you need millions of lightweight concurrent flows, web servers handling millions of connections, stream processors handling millions of events per second, an M:N model can be vastly more efficient than one OS thread per flow.

Chapter 3: Inside the CPU: Cores, Caches and Coherence

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

CPU Cores and Hardware Threads

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

The Memory Hierarchy and Cache Lines

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

MESI and Cache Coherence Protocols

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

NUMA Architectures and Their Consequences

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Memory Buses and Interconnect Latencies

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Chapter 4: False Sharing, Contention and Cache Thrashing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

The Anatomy of False Sharing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Detecting False Sharing in Practice

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Memory Allocation Under Concurrency

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Cache Thrashing and Its Symptoms

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Designing for Cache-Line Awareness

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Chapter 5: How CPUs Reorder Memory Access

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Why CPUs Reorder Instructions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

The x86-64 Memory Model

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

ARM and Weakly Ordered Architectures

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Compiler Reordering vs Hardware Reordering

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

What This Means for Correct Code

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Chapter 6: The Happens-Before Relation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Defining Happens-Before

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Program Order and Sequential Execution

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Synchronization Order and Atomic Operations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Visibility Guarantees and Their Limits

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Reasoning About Correctness with Happens-Before

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Chapter 7: Consistency Models: From Sequential to Linearizable

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Sequential Consistency and Its Promise

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Why Sequential Consistency Is Expensive

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Linearizability and Its Real-Time Guarantees

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Relaxed Consistency and When It Is Safe

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Choosing the Right Consistency Model

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Chapter 8: Atomic Primitives and Memory Ordering

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Atomicity and What It Actually Guarantees

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Sequentially Consistent Atomics

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Acquire and Release Semantics

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Relaxed Atomics and Their Danger Zones

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Memory Fences and Barrier Instructions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Chapter 9: Compare-and-Swap and Read-Modify-Write Operations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

How Compare-and-Swap Works

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

CAS at the Hardware Level

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Fetch-and-Add and Other RMW Operations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

The ABA Problem in Detail

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Tagged Pointers and Version Counters

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Chapter 10: Language-Level Memory Models: C/C++, Java, Rust, Go

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

The C11/C++11 Memory Model

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Java's Happens-Before Semantics

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Rust's Approach to Concurrency

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Go's Simpler Memory Guarantees

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Mapping Between Language Models

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Chapter 11: Progress Guarantees and Fairness

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

What It Means for a System to Make Progress

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Lock-Based (Blocking) Algorithms

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Lock-Free Algorithms

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Wait-Free Algorithms

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Fairness and Starvation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Priority Inversion

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Deadlock, Livelock and Starvation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Choosing Progress Guarantees

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Chapter 12: Mutexes and Futexes

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

What a Mutex Actually Does

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Futexes: How Modern Mutexes Work

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Spinlocks

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Recursive Mutexes

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Mutex Performance and Scalability

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Chapter 13: Read-Write Locks, Barriers and Condition Variables

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Read-Write Locks

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Condition Variables

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Barriers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Semaphores

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Monitors

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

When to Use Which Primitive

Chapter 14: Specialized Locks: Ticket Locks, MCS Locks and Seqlocks

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Why Specialized Locks Exist

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Ticket Locks

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

MCS Locks

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Q-locks and CLH Locks

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Seqlocks

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Choosing a Lock for High Contention

Chapter 15: Lock-Free Programming Fundamentals

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

When Lock-Free Is Worth It

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

The CAS Loop Pattern

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Lock-Free vs Wait-Free vs Blocking

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

ABA in Lock-Free Structures

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Memory Ordering in Lock-Free Code

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Chapter 16: Reference Counting and Atomic Pointers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Why Reference Counting Matters in Concurrency

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Atomic Reference Counting

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Optimized Reference Counting

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Control Blocks and Shared State

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Reference Counting and Lock-Free Structures

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Chapter 17: Lock-Free Stacks, Queues and Ring Buffers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

The Treiber Stack

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

The Michael-Scott Queue

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Single-Producer Single-Consumer Ring Buffer

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Multi-Producer Multi-Consumer Ring Buffer

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

When to Use Which Structure

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Chapter 18: Memory Reclamation: Hazard Pointers, EBR and RCU

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

The Memory Reclamation Problem

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Hazard Pointers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Epoch-Based Reclamation (EBR)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

RCU (Read-Copy-Update)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Comparing Memory Reclamation Techniques

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Chapter 19: Concurrent Hash Maps and Scalable Data Structures

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Why Hash Maps Are Hard in Concurrency

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Lock-Striped Hash Map

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Java's ConcurrentHashMap

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

C++ and the Lack of Standard Concurrent Containers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Immutable and Copy-on-Write Maps

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

When to Use Which Approach

Chapter 20: Diagnosing Data Races and Visibility Bugs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Why Data Races Are Insidious

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Symptoms of Data Races

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

ThreadSanitizer (TSan)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Java ThreadSanitizer Alternatives

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Rust's Compile-Time Prevention

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Go's Race Detector

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Lockset Algorithms

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Static Analysis for Races

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Best Practices for Race Detection

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Chapter 21: Debugging Deadlocks, Livelocks and Starvation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Deadlock Detection in Production

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Thread Dump Analysis

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Deadlock Detection Algorithms

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Preventing Deadlock

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Livelock Detection

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Starvation Detection

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Tools for Debugging Liveness Bugs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Chapter 22: Concurrency Testing and Verification

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Why Normal Testing Is Not Enough

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Stress Testing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Property-Based Testing for Concurrency

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Model Checking

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Deterministic Concurrency Testing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Fuzzing for Concurrency Bugs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Verification for Lock-Free Algorithms

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Testing Checklist

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Chapter 23: Performance Profiling and Analysis

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Measuring What Matters

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Flame Graphs for Concurrent Code

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Per-Thread and Per-Core Analysis

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Cache Performance Analysis

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Lock Contention Analysis

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Context Switch and Scheduler Analysis

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Tools Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Chapter 24: Benchmarking Methodology for Concurrent Systems

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Why Benchmarking Concurrent Code Is Hard

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Controlling Environmental Variables

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Measuring Distributions, Not Averages

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Scalability Testing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Avoiding Benchmark Artifacts

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Comparing Synchronization Strategies

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Benchmarking Checklist

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Chapter 25: Concurrency Design Patterns and Trade-Offs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Thread Pools

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Work Stealing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Futures and Promises

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Message Passing and Channels

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Sharding and Partitioning

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Choosing the Right Pattern

Chapter 26: Production Architectures and Case Studies

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

High-Throughput Web Servers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Database Concurrency: MVCC and Latching

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Redis: Single-Threaded with Clustering

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Kafka: Partitioned Log with Concurrent Consumers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Go Runtime: M:N Scheduling

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Lessons from Production

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Chapter 27: Advanced Topics and Language-Specific Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

C++ Concurrency Best Practices

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Rust Concurrency Best Practices

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Go Concurrency Best Practices

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.

Java Concurrency Best Practices

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/theartofconcurrentprogramming.