Java Concurrency in Practice

Chapter 1: Introduction

- ***Safety*** means "nothing bad ever happens" while **liveness** means "something good eventually happens."
- Compromising safety often means compromising correctness. An example is a race condition between multiple threads.
- A liveness failure in a single-threaded program can be an infinite loop. In a multithreaded program, it could be deadlock, starvation, or livelock.
- While liveness means that something good **eventually** happens, performance means that it will happen **quickly**.
- Context switches have significant costs, including saving and restoring execution context, less of locality, and spending CPU time on scheduling threads instead of running them.

Chapter 2: Thread Safety

- Writing thread-safe code is, at its core, about managing access to shared, mutable state.
- Whenever more than one thread accesses a given state variable, and one of them might write to it, they all must coordinate their access to it using synchronization.
- A broken program shares mutable state without synchronization. Either don't share the state, make it immutable, or use synchronization upon every access.
- It is far easier to design a class to be thread-safe than to retrofit it for safety later.
- The same object-oriented techniques that help you write well-organized, maintainable classes -- such as encapsulation and data hiding -- can also help you create thread-safe classes.
- A program that consists entirely of thread-safe classes may not be thread-safe, and a thread-safe program may contain classes that are not thread-safe.

2.1: What is thread safety?

- A class is thread-safe if it behaves correctly when accessed from multiple threads, regardless of their scheduling or interleaving of execution, and with no additional synchronization or coordination by the calling code.
- Thread-safe classes encapsulate any needed synchronization so that clients need not provide their own.

2.2: Atomicity

- A race condition occurs when the correctness of a computation depends on the relative timing or interleaving of multiple threads.
- The most common type of race condition is *check-then-act*, where an observation could have become invalid between the time you observed it and the time you acted on it, causing a problem.
- Read-modify-write operations require knowing a previous value and ensuring that no one else changes or uses that value while you are in mid-update.
- Check-then-act and read-modify-write sequences are compound actions, or sequences that must be executed atomically in order to remain thread-safe.

2.3: Locking

- Mutual exclusion locks, or mutexes, means that at most one thread may own a lock. If thread A attempts to acquire a lock by thread B, it blocks until B releases it.
- Reentrancy means that locks are acquired on a per-thread rather than per-invocation basis. Each lock is associated with an acquisition count and an owning thread.

2.4: Guarding state with locks

- For each mutable state variable that may be accessed by more than one thread, all accesses to that variable must be performed by the same lock held. The variable is *guarded* by the lock.
- When a class has invariants that involve more than one state variable, we must ensure that each variable participating in the invariant must be guarded by the same lock.

2.5: Liveness and performance
- Acquiring and releasing a lock has some overhead, so do not break down guarded blocks too far, even if this would not compromise atomicity.
- When implementing a synchronization policy, resist the temptation to prematurely sacrifice simplicity (potentially compromising safety) for the sake of performance.
- Avoid holding locks during lengthy computations or operations at risk of not completing quickly, such as when performing disk or network I/O.

Chapter 3: Sharing Objects

3.1: Visibility
- To ensure visibility of memory writes across threads, you must use synchronization.
- There is no guarantee that operations in one thread will be performed in the order given by the program, as long as the reordering is not detectable from within **that** thread.
- Because of reordering, attempts to reason about the order in which memory actions "must" happen in insufficiently synchronized multithreaded programs will almost certainly be incorrect.
- Unless synchronization is used every time a variable is accessed, it is possible for a thread to read a stale value for that variable.
- Stale data can cause serious and confusing failures such as unexpected exceptions, corrupted data structures, inaccurate computations, and infinite loops.
- **Out-of-thin-air safety** is when a variable reads a thread without synchronization, it reads a value that was actually written by another thread instead of some random value.
- With the Java Memory Model, the values of variables that were visible to thread A prior to releasing a lock are guaranteed to be visible to thread B upon acquiring the same lock.
- Variables marked with the `volatile` keyword are not reordered with other memory operations, and are not put in caches where they are hidden from other processors.
- A `volatile` variable can be used for a completion, interruption, or status flag, but the semantics are not strong enough to make the increment operation atomic, for example.

3.2: Publication and escape
- **Publishing** an object means making it available to code outside of its current scope. An object that is published when it should not have been is said to have **escaped**.
- Any object that is **reachable** from a published object by following some chain of nonprivate field references and method calls has also been published.
- Once an object escapes, you have to assume that another class or thread may, maliciously or carelessly, misuse it. This is a compelling reason to use encapsulation.
- An object is in a predictable, consistent state only after its constructor returns, so publishing an object (via `this`) from within its constructor can publish an incompletely constructed object.
- You can avoid this improper construction by using a private constructor and a public static factory method.

3.3: Thread confinement
- If data is only accessed by a single thread, then no synchronization is needed. This technique is called **thread confinement**.
- **Ad-hoc thread confinement** is when the responsibility for maintaining thread confinement falls entirely on the implementation.
- Thread confinement is often a consequence of deciding to implement a particular subsystem, such as the GUI, as a single thread. The simplicity benefit of such a system outweighs the fragility of ad-hoc thread confinement.
- **Stack confinement** is a special case of thread confinement in which an object can only be reached through local variables. Local variables are intrinsically confined to the executing thread.

3.4: Immutability
- Immutable objects are simple, because they can only have one state. Immutable objects are also safe, because you can freely share and publish them without the need to make defensive copies.
- An object is immutable if its state cannot be modified after construction, all its fields are `` `final` ``, and it is properly

constructed, i.e. the `this` reference does not escape during construction.
- Whenever a group of related data items but be acted upon atomically, consider creating an immutable holder class for them.

3.5: Safe publication

- Simply storing a reference to an object into a public field is not enough to publish that object safely. Improper publication allows another thread to observe a partially constructed object.
- However, immutable objects can be used safely by any thread without additional synchronization, even when synchronization is not used to publish them.
- To publish an object safely, both the reference to the object and the object's state must be made visible to other threads at the same time.
- Objects that are not technically immutable, but whose state will not be modified after publication, are called **effectively immutable**. Such objects, when safely published, can be used safely by any thread without additional synchronization.
- While effectively immutable objects must be safely published, mutable objects must be safely published, and must be either thread-safe or guarded by a lock.
- Many concurrency errors stem from failing to understand the "rules of engagement" for a shared object. When you publish an object, document how it should be accessed.

Chapter 4: Composing Objects

4.1: Designing a thread-safe class

- Designing a thread-safe class requires identifying the invariants that constrain the state variables, and establishing a policy for managing concurrent access to them.
- The smaller the **state space** of an object or variable, the easier it is to reason about. Use `final` fields to reduce the state space.
- Operations with state-based preconditions are called ***state-dependent***.
- Ownership implies control, but once you publish a reference to a mutable object, you no longer have exclusive control, but at best "shared ownership."

4.2: Instance confinement

- Encapsulating data within an object confines access to the data to the object's methods, making it easier to ensure that the data is always accessed with the appropriate lock held.
- Instance confinement also allows different state variables to be held by different locks.
- Confined objects can also escape by publishing other objects such as iterators or inner class instances that may indirectly publish the confined objects.
- Using a private lock prohibits client code from acquiring it, whereas a publicly accessible lock allows client code to participate in its synchronization policy, perhaps incorrectly.

4.3: Delegating thread safety

- A class with multiple independent thread-safe state variables and no operations that have any invalid state transitions can delegate thread safety to the underlying state variables.
- If a state variable is thread-safe, does not participate in any invariants that constrain its value, and has no prohibited state transitions for any of its operations, then it can be safely published.

4.4: Adding functionality to existing thread-safe classes

– Extending a class to support a thread-safe operation is more fragile than adding code directly to the class, as its synchronization policy is now distributed over multiple source files.
- Just as subclassing violates encapsulation of implementation, client-side locking violates encapsulation of synchronization policy.

4.5: Documenting synchronization policies

- Document a class's thread safety guarantees for its clients; document its synchronization policy for its maintainers.
- If you want clients to be able to create new atomic operations on your class, you must document which locks they should acquire to do so safely.

- If you must guess whether a class is thread-safe, improve the quality of your guess by interpreting the specification by someone who must implement it versus someone who will merely use it.

Chapter 10: Avoiding Liveness Hazards

10.1: Deadlock

- When thread A holds lock L and tries to acquire lock M, but at the same time thread B holds lock M and tries to acquire L, this is deadlock, or ***deadly embrace***.
- When a database system detects that a set of transactions has deadlocked by searching the is-waiting-for graph for cycles, it picks a victim and aborts that transaction, thereby releasing its held locks.
- A program will be free of lock-ordering deadlocks if all threads acquire the locks they need in a fixed global order. Sometimes we must induce this ordering.
- Invoking an alien method with a lock held is asking for liveness trouble. That method might risk deadlock by acquiring other locks, or block for an unexpectedly long time and stall other threads on the held lock.
- Calling a method with no locks held is called an ***open call***, and classes that rely on open calls are more well-behaved and composable than classes that make calls with locks held.
- Resource deadlocks occur when thread A holds resource X and tries to acquire resource Y, but at the same time thread B holds resource Y and tries to acquire resource X.

10.2: Avoiding and diagnosing deadlocks

- Try acquiring locks with a timeout. By using a timeout that is much longer than you expect acquiring the lock to take, you can regain control when something unexpected happens.
-Thread dumps not only include a stack trace for each running thread, but locking information such as which locks are held by a thread, and which lock a thread is waiting to acquire.

10.3: Other liveness hazards

- ***Starvation*** is when a thread is perpetually denied access to resources it needs in order to make progress; the most commonly starved resource is CPU cycles.
- ***Livelock*** is a liveness failure in which a thread, while not blocked, still cannot make progress because it keeps retrying an operation that will always fail.
- Livelock can occur when multiple cooperating threads change their state in response to the others in such a way that no thread can ever make progress. The solution is to introduce randomness into the retry mechanism.

Chapter 11: Performance and Scalability

11.1: Thinking about performance

- Improving performance means doing more work with fewer resources. When performance of an activity is limited by availability of a particular resource, it is ***bound*** by that resource.
- Using concurrency to achieve better performance means using the processing resources we have more effectively, and enable a program to exploit additional processing resources that become available.
- ***Scalability*** describes the ability to improve throughput or capacity when additional computing resources (such as CPUs, memory, storage, or I/O bandwidth) are added.
- The "how much" aspects like scalability, throughput, and capacity are concern for server applications. The "how fast" aspects like service time or latency are concern for client applications.
- Most optimizations are premature because they are often undertaken before a clear set of requirements is available.
- Make it right, then make it fast. And if attempting to make it fast, measure. Don't guess.

11.2: Amdahl's law

- Amdahl's law describes how much a program can theoretically be sped up by additional computing resources, based on the proportion of parallelizable to serial components.
- If F is the faction of the calculation that must be executed serially, then on a machine with N processors, we can achieve a speedup of most: 1/(F+(1-F)/N).
- When evaluating an algorithm, thinking "in the limit" about what would happen with hundreds or thousands of processors can offer some insight into where scaling limits might appear.

11.3: Costs introduced by threads

- When a new thread is switched in, the data it needs is unlikely to be in the local processor cache, and so a context switch causes a flurry of cache misses and runs a little slower at first.
- Schedulers give each runnable thread a certain minimum time quantum, thereby amortizing the cost of the context switch and its consequences over more interrupted execution time.
- A program that does more blocking has more of its threads suspended and switched out. The program therefore incurs more context switches, increasing scheduling overhead and reducing throughput.
- Special instructions called ***memory barriers*** can flush or invalidate caches and flush hardware write buffers. They inhibit compiler optimizations; most operations cannot be reordered with them.
- ***Lock elision*** optimizes away lock acquisitions. ***Lock coarsening*** merges together adjacent blocks holding the same lock, reducing synchronization overhead and helping the optimizer.
- When a lock is contended, the losing threads must block. This can be implemented either by ***spin-waiting*** or by ***suspending*** the blocked thread through the operating system.

11.4: Reducing lock contention

- Two factors influence the likelihood of contention for a lock: How often that lock is requested, and how long it is held once acquired.
- ***Lock splitting*** and ***lock striping*** involve using separate locks to guard multiple independent state variables previously guarded by a single lock.
- Splitting a lock into two offers the greatest possibility for improvement when a lock is experiencing moderate but not heavy contention.
- Lock striping extends lock splitting by partitioning locking on a variable-sized set of independent objects. But locking the collection for exclusive access is more difficult and costly.
- If your class has a small number of hot fields that do not participate in invariants with other variables, then replacing them with atomic variables may improve scalability.
- Tools like `vmstat` or `mpstat` can show whether your application is CPU-bound, while tools like `iostat` or `perfmon` can show whether your application is I/O-bound.
- The tool `vmstat` has a column reporting the number of threads that are runnable but not currently running because a CPU is not available.