**Concurrency problems**

**What is Concurrency?**

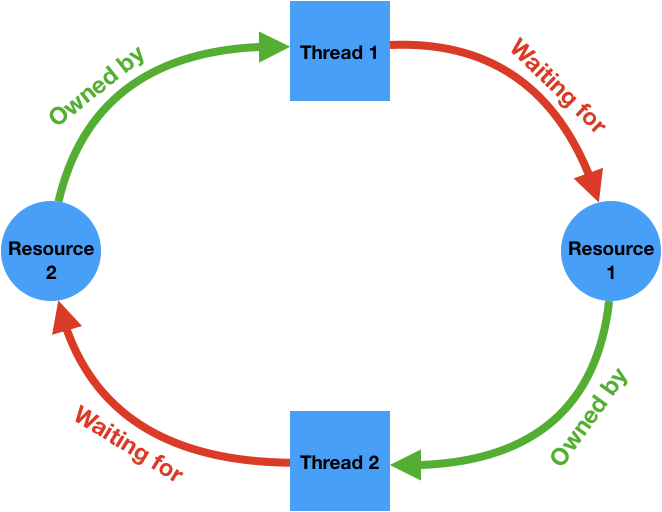
[Concurrency](https://medium.com/javarevisited/6-multithreading-and-concurrency-books-every-java-programmer-should-read-b6a08d2aae54)refers to executing several computations simultaneously. Every programming language has various strategies for managing concurrency. Running multiple simultaneous threads is a simple task as long as they do not interact with mutable shared objects.

Concurrency is important because it allows your application to accomplish more work in less time. Mutable shared objects are objects that are shared or accessible by multiple threads but can also be changed by multiple threads. Ideally, any objects that are shared between threads will be [immutable](https://medium.com/javarevisited/how-to-create-an-immutable-list-list-and-map-in-java-5ac1254c128), and hence immutable shared objects do not pose issues to multithreaded code.

Java manages thread concurrency using [*synchronization*](https://javarevisited.blogspot.com/2017/02/10-java-wait-notify-locking-and-synchronization-Interview-Questions-Answers.html). Each object in Java has a lock and when a thread wants to execute the code in a synchronized block it must first obtain the object’s lock. If the [lock](https://javarevisited.blogspot.com/2014/10/how-to-use-locks-in-multi-threaded-java-program-example.html)is not available then it means that another thread already has the lock so this thread must wait for the lock to be released. [Synchronization](https://javarevisited.blogspot.com/2011/04/synchronization-in-java-synchronized.html)can lead to all kinds of functional and performance issues.

There are 3 major issues we have to check in terms of concurrency. They are

**1. Thread Deadlocks**

[[](https://javarevisited.blogspot.com/2018/08/how-to-avoid-deadlock-in-java-threads.html)](https://javarevisited.blogspot.com/2018/08/how-to-avoid-deadlock-in-java-threads.html)

Deadlock

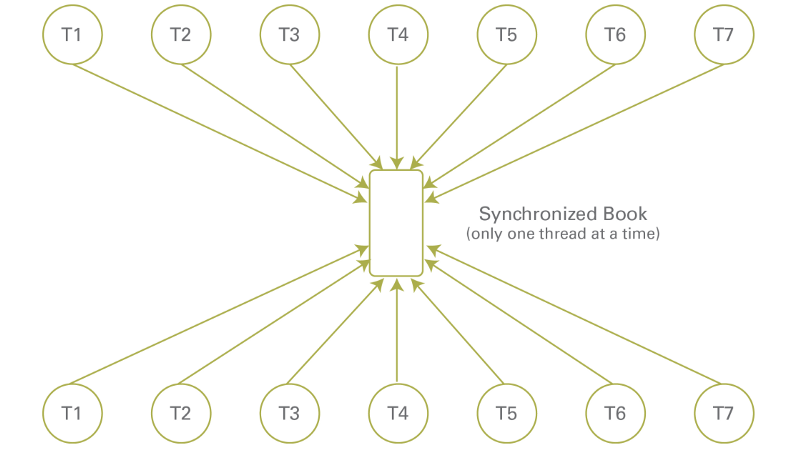
[Deadlocks](https://enlear.academy/how-to-avoid-a-deadlock-while-writing-concurrent-programs-java-example-988bb07db25f) occur when two or more threads need multiple shared resources to complete their task and they access those resources in a different order or a different manner. When a method or a block of code is synchronized, the executing thread obtains the lock for the object upon which the code is synchronized, executes the [synchronized code](https://javarevisited.blogspot.com/2011/05/wait-notify-and-notifyall-in-java.html), and then relinquishes that lock. If a second thread attempts to execute the synchronized code while the first thread has the lock then the second thread “waits” until the lock is available.

When your application has a deadlock in it, your [JVM](https://medium.com/javarevisited/7-best-courses-to-learn-jvm-garbage-collection-and-performance-tuning-for-experienced-java-331705180686?source=---------8------------------)will eventually exhaust all or most of its threads. The application will appear to be accomplishing less and less work, but the CPU utilization of the machine on which the application is running will appear underutilized.

The deadlock issue is very serious and will eventually cause the application to stop processing business transactions. Even worse, the only way to resolve the issue is to restart the [JVM](https://javarevisited.blogspot.com/2019/04/top-5-courses-to-learn-jvm-internals.html), which takes it out of availability to service your users. When you have multiple instances of your application, it gets even tougher to analyze.

Finally, because deadlocks are the result of race conditions (multiple threads competing for resources and typically using them for a very short period of time) they are very difficult to reproduce in a non-production environment and hence very difficult to troubleshoot.

**2. Thread Gridlocks**

[[](https://javarevisited.blogspot.com/2014/07/top-50-java-multithreading-interview-questions-answers.html)](https://javarevisited.blogspot.com/2014/07/top-50-java-multithreading-interview-questions-answers.html)

Gridlock

**Gridlock** is a similar issue to deadlock. An application under heavy load is like a freeway during rush hour — things are moving a little slower than usual, but for the most part, everything is working fine.

If an application is **over-synchronized**, then you have essentially merged all the lanes of your freeway down to one, resulting in a lot of slow and stalled threads. [Too much synchronization is an over-kill](https://javarevisited.blogspot.com/2015/05/top-10-java-multithreading-and.html)and in such applications, there will not be any issues on the low-load. If the load increases, the application eventually slows down and this is an indication of Thread Gridlock.

Hence it is better to avoid synchronization if you can or use it less with precaution.

**3. Thread Livelock**

Livelock is another concurrency problem that is similar to deadlock. In livelock, two or more threads keep on transferring states between one another instead of waiting infinitely. Consequently, the threads are not able to perform their respective tasks.

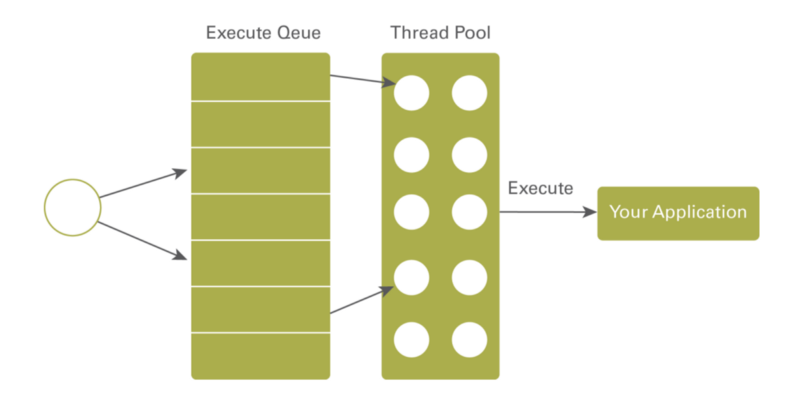
A good example of livelock is a messaging system where, when an exception occurs, the message consumer rolls back the transaction and puts the message back to the head of the [queue](https://javarevisited.blogspot.com/2014/06/synchronousqueue-example-in-java.html). Then the same message is repeatedly read from the queue, only to cause another exception and be put back on the queue. The consumer will never pick up any other message from the queue.

This is a real example I faced recently. My [Spring Boot consumer](https://javarevisited.blogspot.com/2022/03/spring-boot-kafka-example-single-and-multiple-consumers.html) reads the message from the AWS SQS queue and if any issue in the consumer code, an exception is thrown and the message is sent back to the queue. Again the message is processed by all the application instances for 4 days (default retention time of the message in the SQS queue). It goes in a cyclic loop and has caused too many CPU spikes and memory issues resulting in the application shut down.

**3.1 Avoiding Livelock**

To avoid livelock, we have to check the condition that throws the exception and fix it. Instead of throwing exceptions back to the queue, we can log the error message and then exit the consumer gracefully. Later, we can analyze the error message to check the issue which will help us to achieve the goal.

**4. Thread Pool Sizing**

[[](https://javarevisited.blogspot.com/2013/07/how-to-create-thread-pools-in-java-executors-framework-example-tutorial.html)](https://javarevisited.blogspot.com/2013/07/how-to-create-thread-pools-in-java-executors-framework-example-tutorial.html)

Thread Pool

If your application is running in an application server or web container, it will have a thread pool configured to control how many requests your application can concurrently process. These threads are different from the threads we create in our application.

The [creation of threads](https://javarevisited.blogspot.com/2012/03/difference-between-start-and-run-method.html) is a costly operation and hence the application servers will create them when the server is started. In Java, threads are mapped to system-level threads, which are the operating system’s resources. If we create threads uncontrollably, we may run out of these resources quickly.

When the server receives a request from a socket listener, it places the request in an execution queue and then returns to listen for the next request to arrive on the socket. The execution queue is served by a thread pool. When a thread is available in the thread pool, a request is removed from the execution queue and passed to a thread for processing. This particular thread is mapped to the request and it takes care of serving the request.

The thread executes the appropriate business transaction in your application code. When the thread completes processing the thread is returned to the thread pool and will be available to process another request.

The configuration of the size of that thread pool is very important to the performance of your application. If the thread pool is sized too small then your requests are going to wait but if the thread pool is sized too large then too many threads are going to execute concurrently and take all of the machine’s processing resources.

When there are too many threads and the machine spends too much time context switching between threads, the threads will be starved of CPU cycles and will take longer to complete. You have a finite number of cores in your CPU and if threads need computing power then there is only so much processing power to go around. The behavior of your application will dictate the optimal size of your thread pools.

**4.1 Metrics**

The key is to look at two metrics

1. Thread pool utilization

2. CPU utilization

If your thread pool utilization is high and your CPU utilization is moderate then your thread pool is configured too small. If your thread pool utilization is moderate, but your CPU utilization is high then your thread pool is configured as too large.

For [Spring Boot](https://medium.com/javarevisited/10-best-java-microservices-courses-with-spring-boot-and-spring-cloud-6d04556bdfed), [Tomcat](https://javarevisited.blogspot.com/2016/12/how-to-increase-heap-memory-of-apache.html)is the default server and it creates 200 threads and places them in the ThreadPool on server startup. We can configure these numbers in the application.properties files. Similarly, for every application, we can tune the thread pool count by trial and error and arrive at the optimized number.

**4.2 Formula to calculate the thread pool size**

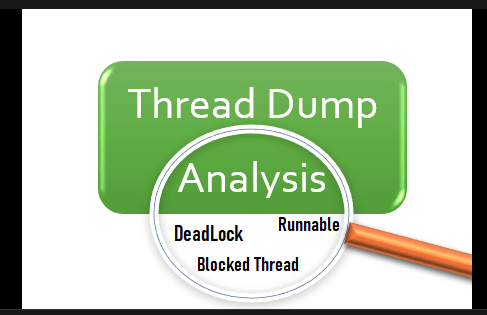
Number of threads = Number of Available Cores \* (1 + Wait time / Service time)int numOfCores = Runtime.getRuntime().availableProcessors();

**Waiting time** — is the time spent waiting for IO-bound tasks to complete, i.e waiting for an HTTP response from a remote service.

**Service time** — is the time spent being busy, say processing the HTTP response, marshaling/unmarshaling, any other transformations, etc.

Wait time / Service time — this ratio is often called the *blocking coefficient*.

**Thread Dump Analysis**

[[](https://javarevisited.blogspot.com/2011/09/javalangoutofmemoryerror-permgen-space.html)](https://javarevisited.blogspot.com/2011/09/javalangoutofmemoryerror-permgen-space.html)

Thread Dump Anaysis

A thread dump provides a snapshot of the current state of a running Java process. By analyzing the thread dump, we can find out the above issues related to the concurrency. jVisualVM is a default profiler that comes up with the [JDK](https://javarevisited.blogspot.com/2016/06/maven-eclipse-error-no-compiler-is-no-JRE-JDK-installed.html). And there are many other Java Profilers we can use to analyze the Thread Dumps.

In you are using Spring Boot, it is easier to get the Thread Dumps of the application

**application.properties — expose all endpoints**

management.endpoints.web.exposure.include=\*

**application.properties — expose thread dump endpoints**

management.endpoints.web.exposure.include=threaddump

**Url to get thread dump**

http://localhost:8080/actuator/threaddump