Difference between Lock, synchronized, and Semaphore in Java

JAVINPAUL

MAR 23, 2024 • PAID

Hello guys, how are you doing? In the past few episodes, I have shared popular Java interview questions like How ConcurrentHashMap work in Java? Why String is Immutable and What is the difference between List, List<Object>, and List<?> Today I am going to share another popular Java interview question, what is the difference between synchronized, Lock, and Semaphore in Java?

My goal is to share two Java interview questions every week so that you are always ready to take your next Java interview, rather than doing last-minute preparation or losing your first few interviews because you are not warmed up. If your goal is the same then please consider getting a paid subscription as these questions will only be for paid subscribers.

Coming back to the topic, In concurrent programming, managing access to shared resources is paramount to ensuring correctness and efficiency. In Java, developers have several mechanisms for synchronizing access among multiple threads. Among the most commonly used are `synchronized`, `Lock`, and `Semaphore`. Each of these mechanisms offers distinct features and trade-offs, catering to different concurrency scenarios.

For example, `synchronized` is a language-level construct in Java used for mutual exclusion, `Lock` provides more flexibility with explicit locking and unlocking, while `Semaphore` allowing controlling access to shared resources through a set of permits, enabling multiple threads to access simultaneously up to a defined limit.

Later we will also see a real-world example of where exactly you can use synchronized, Lock and Semaphore in Java.

In this episode, we delve into the nuances of `synchronized`, `Lock`, and `Semaphore` in Java. We'll explore their functionalities, differences, and use cases, providing insights to help developers make informed decisions when designing concurrent applications.

Difference between Synchronized, Lock, and Semaphore in Java

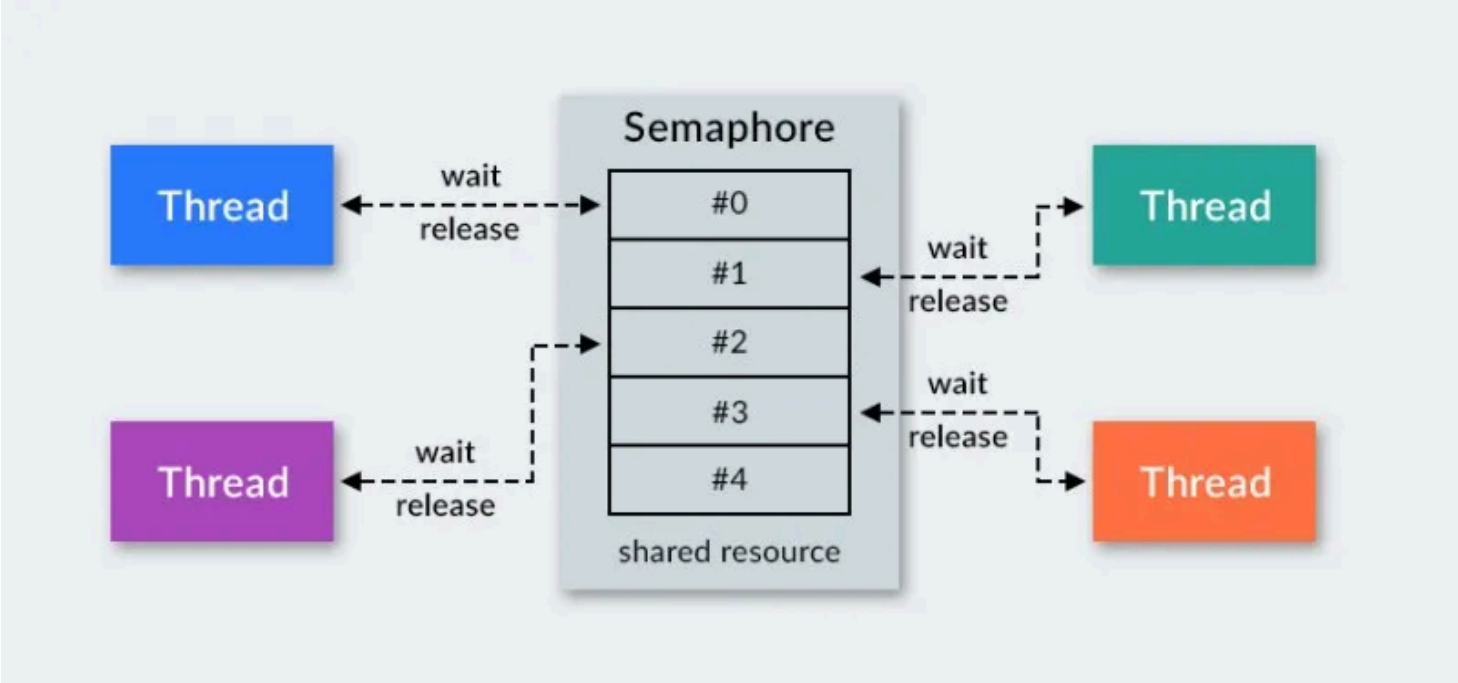
In Java, `synchronized`, `Lock`, and `Semaphore` are all mechanisms used for coordinating access to shared resources among multiple threads. However, they differ in their functionality and usage:

1. `synchronized`:
   - `synchronized` keyword is used to achieve mutual exclusion and to synchronize access to methods or blocks of code.
   - It ensures that only one thread can execute a synchronized method or block at any given time.
   - It is implemented at the language level, making it simpler to use, but it has limitations, such as not being able to specify a timeout for waiting or supporting non-blocking attempts to acquire a lock.
   - Example:
     ```
     public synchronized void synchronizedMethod() {
     // synchronized code here
     }
     ```
2. `Lock`:
   - `Lock` interface in the `java.util.concurrent.locks` package provides a more flexible way of managing synchronization.
   - It allows explicit locking and unlocking, which enables better control over concurrency.
   - It supports more advanced features like timed waits, interruptible locking, and condition variables.
   - Unlike `synchronized`, locks can be acquired and released in different methods or scopes.
   - Example:
     ```
     Lock lock = new ReentrantLock();
     lock.lock();
     try {
     // critical section
     } finally {
     lock.unlock();
     }
     ```
3. `Semaphore`:
   - `Semaphore` is a concurrency utility in the `java.util.concurrent` package that maintains a set of permits.
   - It allows controlling access to a shared resource through a counter that increments when a thread acquires a permit and decrements when a thread releases a permit.

- Unlike locks, semaphores can permit multiple threads to access a resource simultaneously, up to a defined limit.
- Semaphores are useful in scenarios where a resource has a fixed capacity or when a limited number of threads should access the resource simultaneously.
- Example:

```
Semaphore semaphore = new Semaphore(3); // Allow 3 permits
semaphore.acquire(); // Acquire permit
try {
// critical section
} finally {
semaphore.release(); // Release permit
}
```

Here is also a nice diagram showing how Semaphore can be used to manage shared resource:



Real World Example of Synchronized, Lock, and Semaphore in Java

In order to best understand this concept, we need to think about any real-world scenarios where synchronized, Lock, and Semaphore can be used and what difference they make.

Let's consider a real-world scenario where these synchronization mechanisms could be applied: managing access to a shared database connection pool in a multi-threaded application.

- Imagine a web server handling multiple concurrent requests where each request needs to access a shared database connection pool.

- You can use `synchronized` blocks to ensure that only one thread at a time can acquire a database connection from the pool, preventing race conditions and ensuring thread-safe access as shown below:

```
public synchronized Connection getConnection() {
// Code to acquire a connection from the pool
}
```

Now, for Lock, consider a multi-threaded application where threads need exclusive access to a resource for a significant period of time. In this case, you can use a `ReentrantLock` to manage access to the resource.

Threads can explicitly acquire and release the lock, allowing for more fine-grained control as shown below:

```
private Lock lock = new ReentrantLock();

public void doOperation() {

lock.lock();

try {

// Critical section - access to the resource

} finally {

lock.unlock();

}

}
```

Now for Semaphore, consider a scenario where a limited number of connections are available to access a database. In this case, you can use a `Semaphore` to control access to the database connection pool. The semaphore limits the number of threads that can simultaneously acquire connections, ensuring that the pool doesn't get overwhelmed.

here is the code:

```
private Semaphore semaphore = new Semaphore(MAX_CONNECTIONS);

public Connection getConnection() throws InterruptedException {

semaphore.acquire();
```

```
try {

// Code to acquire a connection from the pool

} finally {

semaphore.release();

}

}
```

In summary, while `synchronized` is simpler to use and suitable for basic synchronization needs, `Lock` and `Semaphore` provide more flexibility and advanced features for managing concurrency in complex scenarios. Choosing the appropriate synchronization mechanism depends on the specific requirements of your application.

Let me know how did you find this article

All the best for your interviews!!

*And, if need help with your interview preparation, you can also check my Java + Spring Interview + SQL Bundle on Gumroad, use code **friends20** to get a 20% discount also*

4 Likes · 1 Restack

← Previous

Next →

1 Comment

Write a comment...

Soma Soma's Substack Mar 23

can we synchronized a static final method in Java?

LIKE REPLY SHARE