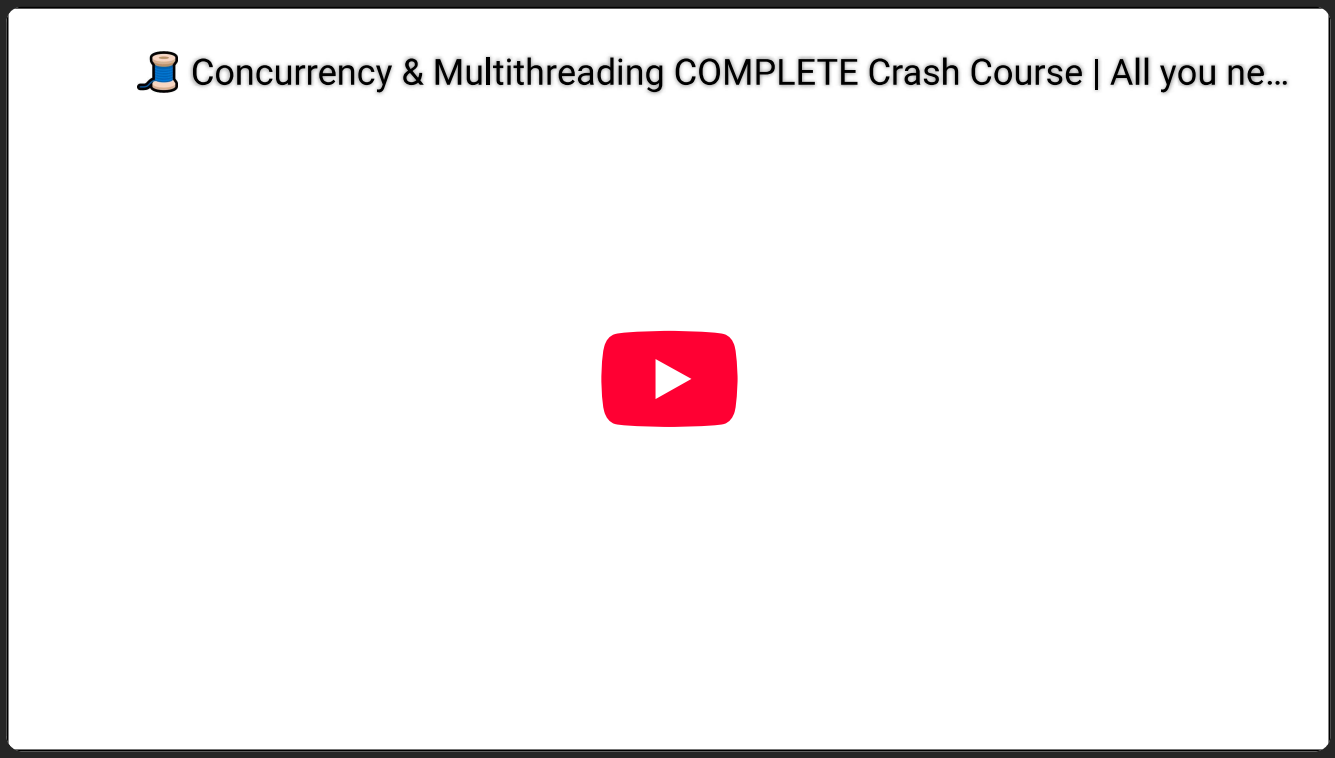
Search articles...

Sign In

Semaphore



Topic Tags:

system design lld

Github Codes Link: https://github.com/aryan-0077/CWA-LowLevelDesignCode

Semaphores in Java: Powerful Concurrency Control

Semaphores are one of the most versatile synchronization mechanisms in concurrent programming. Unlike locks which typically enforce exclusive access, semaphores can control access to a specific number of resources, making them perfect for implementing resource pools, throttling mechanisms, and coordinating thread execution.

What is a Semaphore?

A semaphore is a synchronization primitive that maintains a count of permits. 🎚️ Threads can acquire these permits (decreasing the count) or release them (increasing the count). When a thread attempts to acquire a permit and none are available, the thread blocks until a permit becomes available or until it's interrupted. 🧵🚦

Conceptually, a semaphore has two primary operations:
🔷 acquire(): Obtains a permit, blocking if necessary until one becomes available 🛑
🔷 release(): Returns a permit to the semaphore ✅

Types of Semaphores 🎯

1. Binary Semaphore 🔄

> *A binary semaphore has only two states (0 or 1 permit) and is mainly used to enforce mutual exclusion, similar to a mutex or lock.*

Java

```java
import java.util.concurrent.Semaphore;

public class BinarySemaphoreExample {
    private static final Semaphore mutex = new Semaphore(1); // Binary sem
    public static void main(String[] args) {
        Thread t1 = new Thread(() -> accessCriticalSection("Thread-1"));
        Thread t2 = new Thread(() -> accessCriticalSection("Thread-2"));
        t1.start();
        t2.start();
    }


    private static void accessCriticalSection(String threadName) {
        try {
            System.out.println(threadName + " is attempting to acquire the
            mutex.acquire(); // Acquire the semaphore
            System.out.println(threadName + " acquired the lock.");
            Thread.sleep(1000); // Simulate work in the critical section
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        } finally {
            mutex.release(); // Release the semaphore
            System.out.println(threadName + " released the lock.");
```

```
        }
    }
}
```

Output :

```
Thread-1 is attempting to acquire the lock.
Thread-1 acquired the lock.
Thread-2 is attempting to acquire the lock.
Thread-1 released the lock.
Thread-2 acquired the lock.
Thread-2 released the lock.
```

2. Counting Semaphore 🔢

> *A counting semaphore allows multiple permits, making it suitable for managing access to a pool of resources. It can have any non-negative number of permits.*

Java

```java
import java.util.concurrent.Semaphore;

public class CountingSemaphoreExample {
    private static final Semaphore resourcePool = new Semaphore(3); // Sem

    public static void main(String[] args) {
        for (int i = 1; i <= 5; i++) {
            final int threadNum = i;
            Thread t = new Thread(() -> accessResource("Thread-" + threadN
            t.start();
        }
    }

    private static void accessResource(String threadName) {
        try {
            System.out.println(threadName + " is attempting to acquire a p
            resourcePool.acquire(); // Acquire a permit
            System.out.println(threadName + " acquired a permit.");
            Thread.sleep(2000); // Simulate resource usage
```

```
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            } finally {
                resourcePool.release(); // Release the permit
                System.out.println(threadName + " released the permit.");
            }
        }
    }
```

Output :

```
Thread-1 is attempting to acquire a permit.
Thread-1 acquired a permit.
Thread-2 is attempting to acquire a permit.
Thread-2 acquired a permit.
Thread-3 is attempting to acquire a permit.
Thread-3 acquired a permit.
Thread-4 is attempting to acquire a permit.
Thread-5 is attempting to acquire a permit.
Thread-1 released the permit.
Thread-4 acquired a permit.
Thread-2 released the permit.
Thread-5 acquired a permit.
Thread-3 released the permit.
Thread-4 released the permit.
Thread-5 released the permit.
```

Common Use Cases of Semaphores 🎯

1. Managing Access to a Pool of Resources:
Semaphores are ideal for controlling access to a limited number of resources, such as database connections, file handlers, or thread pools. 🗂️🔒

Example :

Java

```
Semaphore resourcePool = new Semaphore(5); // 5 permits for 5 resources
// Threads acquire permits to access the shared resource and release them af
```

2. **Implementing Producer-Consumer Pattern**: Semaphores can synchronize producer and consumer threads by using separate semaphores to track empty and filled slots in a buffer. 🛒📦

Example:

Java

```java
Semaphore emptySlots = new Semaphore(bufferSize); // Track empty slots
Semaphore filledSlots = new Semaphore(0); // Track filled slots
```

3. **Controlling Concurrency Levels:** When performing parallel computations, semaphores can limit the number of threads running concurrently to avoid overwhelming the system. 🧵⚖️

Example:

Java

```java
Semaphore maxThreads = new Semaphore(10); // Restrict to 10 threads at a tim
```

4. **Enforcing Mutual Exclusion (Binary Semaphore):** Binary semaphores act like mutexes to ensure that only one thread accesses a critical section at a time. 🚧🔒

Example:

Java

```java
Semaphore mutex = new Semaphore(1); // Single permit for mutual exclusion
```

Interview Questions 🎯

1. What's the difference between a Semaphore and a Lock? 🎯🔒
Answer: A Lock allows only one thread to access a resource at a time (mutual exclusion), while a Semaphore can allow a specified number of threads to access resources concurrently. A Lock is owned by a specific thread that must release it, whereas Semaphore permits can be acquired and released by different threads. Locks support multiple condition variables, while Semaphores work on a simpler permit-based model. 🔄

Java

```java
public class SemaphoreVsLockExample {
    private final Semaphore semaphore = new Semaphore(3); // Allows up to
    private final Lock lock = new ReentrantLock();
    // Using Semaphore
    public void accessWithSemaphore() {
        try {
            semaphore.acquire(); // Acquire a permit; up to 3 threads can
            System.out.println(Thread.currentThread().getName() + " access
            Thread.sleep(1000); // Simulate work
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            System.out.println(Thread.currentThread().getName() + " releas
            semaphore.release(); // Release the permit
        }
    }
    // Using Lock
    public void accessWithLock() {
        lock.lock(); // Acquire the lock (only one thread can hold it)
        try {
            System.out.println(Thread.currentThread().getName() + " access
            Thread.sleep(1000); // Simulate work
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            System.out.println(Thread.currentThread().getName() + " unlock
            lock.unlock(); // Release the lock
        }
    }

    public static void main(String[] args) {
        SemaphoreVsLockExample example = new SemaphoreVsLockExample();
        // Create and start threads with descriptive names
        for (int i = 1; i <= 5; i++) {
            Thread semaphoreThread = new Thread(example::accessWithSemapho
            Thread lockThread = new Thread(example::accessWithLock, "LockT
            semaphoreThread.start();
            lockThread.start();
        }
```

```
    }
}
```

Output : (Output can vary with each run because of the nature of thread scheduling) :

```
SemaphoreThread-1 accessing resource with Semaphore
    SemaphoreThread-2 accessing resource with Semaphore
    SemaphoreThread-3 accessing resource with Semaphore
    LockThread-1 accessing resource with Lock
    LockThread-1 unlocking Lock
    SemaphoreThread-1 releasing Semaphore permit
    LockThread-2 accessing resource with Lock
    SemaphoreThread-2 releasing Semaphore permit
    LockThread-2 unlocking Lock
    SemaphoreThread-3 releasing Semaphore permit
    SemaphoreThread-4 accessing resource with Semaphore
    LockThread-3 accessing resource with Lock
    LockThread-3 unlocking Lock
    SemaphoreThread-4 releasing Semaphore permit
    SemaphoreThread-5 accessing resource with Semaphore
    LockThread-4 accessing resource with Lock
    LockThread-4 unlocking Lock
    SemaphoreThread-5 releasing Semaphore permit
    LockThread-5 accessing resource with Lock
    LockThread-5 unlocking Lock
```

2. What happens if a thread calls release() on a semaphore without first calling acquire()?
Answer: In Java's Semaphore implementation, calling release() without a prior acquire() is perfectly legal. It simply increases the permit count beyond its initial value. This behavior can be useful in certain scenarios, such as dynamically increasing the number of available resources. However, this can lead to unexpected behavior if not managed carefully, as it might allow more concurrent access than originally intended.

Java

```java
import java.util.concurrent.Semaphore;
public class SemaphoreReleaseExample {
    private final Semaphore semaphore = new Semaphore(2); // Initially all
    public void accessResource() {
        try {
            semaphore.acquire(); // Acquire a permit (there may be up to 3
            System.out.println(Thread.currentThread().getName() + " acquir
```

```
            Thread.sleep(1000); // Simulate work
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            System.out.println(Thread.currentThread().getName() + " releas
            semaphore.release(); // Release the permit
        }
    }

    public static void main(String[] args) {
        SemaphoreReleaseExample example = new SemaphoreReleaseExample();
        // Intentionally release a permit without acquiring one
        // This increases the available permit count from 2 to 3.
        example.semaphore.release();
        System.out.println("Permit count after extra release: " + example.
        // Start multiple threads with descriptive names to use the semaph
        for (int i = 1; i <= 3; i++) {
            new Thread(example::accessResource, "SemaphoreThread-" + i).st
        }
    }
}
```

Output :

```
Permit count after extra release: 3
    SemaphoreThread-1 acquired semaphore
    SemaphoreThread-2 acquired semaphore
    SemaphoreThread-3 acquired semaphore
    SemaphoreThread-2 released semaphore
    SemaphoreThread-1 released semaphore
    SemaphoreThread-3 released semaphore
```

3. How would you implement a barrier synchronization pattern using semaphores?
Answer: A barrier ensures that no thread can proceed past a certain point until all threads have reached that point. Here's how to implement it with semaphores:

Java

```java
public class SemaphoreBarrierExecutorDemo {
  // A reusable barrier implemented with semaphores
  static class SemaphoreBarrier {
    private final int parties;
    private int count;
    private final Semaphore mutex = new Semaphore(1);
    private final Semaphore barrier = new Semaphore(0);
    public SemaphoreBarrier(int parties) {
      this.parties = parties;
      this.count = parties;
    }

    public void await() throws InterruptedException {
      mutex.acquire();
      count--;
      if (count == 0) {
        // Last thread arrives: release all waiting threads
        barrier.release(parties - 1);
        // Reset barrier state for reuse
        count = parties;
        mutex.release();
      } else {
        // Release mutex so other threads can update the count
        mutex.release();
        // Wait until the last thread releases this thread
        barrier.acquire();
      }
    }
  }

  public static void main(String[] args) {
    final int numThreads = 5;
    final SemaphoreBarrier barrier = new SemaphoreBarrier(numThreads);
    // Create a fixed thread pool with custom thread names
    ExecutorService executor = Executors.newFixedThreadPool(numThreads, ne
      private int counter = 1;
      @Override
      public Thread newThread(Runnable r) {
        Thread t = new Thread(r, "Worker-" + counter);
        counter++;
```

```
41          return t;
42        }
43      });
44      // Submit tasks to the executor
45      for (int i = 0; i < numThreads; i++) {
46        executor.submit(() -> {
47          try {
48            // Phase 1: Some work before reaching the first barrier
49            System.out.println(Thread.currentThread().getName() + " doing ph
50            Thread.sleep((long) (Math.random() * 1000)); // Simulate work
51            System.out.println(
52                Thread.currentThread().getName() + " arrived at barrier afte
53            barrier.await(); // Wait until all threads reach here
54            // Phase 2: This phase begins only after every thread has finish
55            System.out.println(Thread.currentThread().getName() + " starting
56            Thread.sleep((long) (Math.random() * 1000)); // Simulate work
57            System.out.println(Thread.currentThread().getName() + " finished
58            barrier.await(); // Synchronize end of phase 2
59            // Phase 3: The final phase starts after all threads have comple
60            System.out.println(Thread.currentThread().getName() + " starting
61          } catch (InterruptedException e) {
62            Thread.currentThread().interrupt();
63            System.out.println(Thread.currentThread().getName() + " was inte
64          }
65        });
66      }
67      // Initiate an orderly shutdown
68      executor.shutdown();
69      try {
70        if (!executor.awaitTermination(30, TimeUnit.SECONDS)) {
71          System.out.println("Some tasks did not finish in time");
72          executor.shutdownNow();
73        }
74      } catch (InterruptedException e) {
75        System.out.println("Main thread interrupted");
76        executor.shutdownNow();
77      }
78      System.out.println("All tasks completed");
79    }
80  }
```

Output :

```
Worker-1 doing phase 1 work
    Worker-2 doing phase 1 work
    Worker-3 doing phase 1 work
    Worker-4 doing phase 1 work
    Worker-5 doing phase 1 work
    Worker-3 arrived at barrier after phase 1
    Worker-1 arrived at barrier after phase 1
    Worker-2 arrived at barrier after phase 1
    Worker-5 arrived at barrier after phase 1
    Worker-4 arrived at barrier after phase 1
    Worker-2 starting phase 2
    Worker-1 starting phase 2
    Worker-3 starting phase 2
    Worker-5 starting phase 2
    Worker-4 starting phase 2
    Worker-3 finished phase 2
    Worker-1 finished phase 2
    Worker-2 finished phase 2
    Worker-5 finished phase 2
    Worker-4 finished phase 2
    Worker-1 starting phase 3
    Worker-2 starting phase 3
    Worker-3 starting phase 3
    Worker-4 starting phase 3
    Worker-5 starting phase 3
    All tasks completed
```

4. How would you implement a reader-writer lock using semaphores?
Answer: A reader-writer lock allows multiple readers to access a shared resource concurrently,
while ensuring writers get exclusive access. The key principles are:
1. 📚 Multiple readers: Multiple threads can read simultaneously
2. ✍️ Exclusive writers: Only one thread can write at a time
3. 🚫 Mutual exclusion: When a writer is active, no readers are allowed
4. 🔄 Coordination mechanism: First reader blocks writers, last reader unblocks them

Java

```java
public class ReaderWriterLock {
  // Count of active readers.
  private int readerCount = 0;
```

```
4   // Semaphore acting as a mutex for protecting the readerCount variable.
5   private final Semaphore mutex = new Semaphore(1);
6   // Semaphore that allows writers (or the first reader) to acquire exclus
7   private final Semaphore wrt = new Semaphore(1);
8   // Called by a reader to acquire the read lock.
9   public void lockRead() throws InterruptedException {
10    // Acquire the mutex to update the reader count safely.
11    mutex.acquire();
12    readerCount++;
13    // If this is the first reader, acquire the write semaphore to block w
14    if (readerCount == 1) {
15      wrt.acquire();
16    }
17    // Release the mutex so other readers or writers can update the reader
18    mutex.release();
19  }
20
21  // Called by a reader to release the read lock.
22  public void unlockRead() throws InterruptedException {
23    // Acquire the mutex to update the reader count safely.
24    mutex.acquire();
25    readerCount--;
26    // If no readers remain, release the write lock, allowing writers to p
27    if (readerCount == 0) {
28      wrt.release();
29    }
30    mutex.release();
31  }
32
33  // Called by a writer to acquire the write lock.
34  public void lockWrite() throws InterruptedException {
35    // Writers acquire the write semaphore directly.
36    wrt.acquire();
37  }
38
39  // Called by a writer to release the write lock.
40  public void unlockWrite() {
41    wrt.release();
42  }
43
44  // --- Sample usage ---
```

```
  public static void main(String[] args) {
    ReaderWriterLock rwLock = new ReaderWriterLock();
    // Sample reader thread
    Runnable readerTask = () -> {
      try {
        rwLock.lockRead();
        System.out.println(Thread.currentThread().getName() + " is reading
        // Simulate reading time
        Thread.sleep(500);
        System.out.println(Thread.currentThread().getName() + " finished r
        rwLock.unlockRead();
      } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
      }
    };

    // Sample writer thread
    Runnable writerTask = () -> {
      try {
        rwLock.lockWrite();
        System.out.println(Thread.currentThread().getName() + " is writing
        // Simulate writing time
        Thread.sleep(500);
        System.out.println(Thread.currentThread().getName() + " finished w
        rwLock.unlockWrite();
      } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
      }
    };
    // Start sample reader and writer threads
    Thread reader1 = new Thread(readerTask, "Reader-1");
    Thread reader2 = new Thread(readerTask, "Reader-2");
    Thread writer1 = new Thread(writerTask, "Writer-1");
    reader1.start();
    reader2.start();
    writer1.start();
  }
}
```

Output :

```
Reader-1 is reading.
    Reader-2 is reading.
    Reader-2 finished reading.
    Reader-1 finished reading.
    Writer-1 is writing.
    Writer-1 finished writing.
```

Conclusion 💯

Mastering semaphores provides a strong foundation for tackling complex concurrency challenges in your applications. Whether you're designing high-throughput systems, implementing resource pools, or coordinating complex workflows, semaphores offer a versatile tool in your concurrency toolkit.