# 1 Synchronization

Synchronization is a crucial concept that ensures orderly and safe access to shared resources by multiple threads. When multiple threads concurrently access the same resource (such as variables, data structures, or files), synchronization prevents race conditions and data corruption.

**Key points**

- Purpose of Synchronization:
  - Avoid Thread Interference: When multiple threads try to access a shared resource simultaneously, we need to ensure that only one thread accesses it at a time.
  - Maintain Consistency: Synchronization ensures that the shared resource remains consistent and predictable.
- Monitors and Locks:
  - Java implements synchronization using a concept called monitors or locks.
  - Each object in Java is associated with a monitor, which a thread can lock or unlock.
  - Only one thread can own a monitor at a given time.
  - When a thread acquires a lock, it enters the monitor, and other threads attempting to enter the locked monitor are suspended until the first thread exits.
- Thread Synchronization: Coordinates and orders the execution of threads within a multi-threaded program.
  - Two types:
    - Mutual Exclusion: Ensures that threads do not interfere with each other while sharing data. Three ways to achieve mutual exclusion:
      - Synchronized instance methods: Locks the entire method.
      - Synchronized statements: Locks a specific block of code.
      - Synchronized static methods: Applies to static methods or blocks.
    - Cooperation (Inter-thread communication): Allows threads to communicate and coordinate their actions.

## 1.1 Implicit and Explicit Locks

1. **Implicit Locks (Synchronized Blocks)**:
   - Definition: Implicit locks are provided by the `synchronized` keyword in Java.
   - Usage:
     - You can use `synchronized` blocks to protect critical sections of code.
     - When a thread enters a synchronized block, it acquires the lock associated with the object (or class) containing that block.
     - Other threads attempting to enter the same synchronized block are blocked until the lock is released.
   - Advantages:
     - Simplicity: The `synchronized` keyword is built into the language.
   - Limitations:
     - Lack of fine-grained control: You cannot explicitly unlock a synchronized block; it is automatically released when the block exits.
2. **Explicit Locks (java.util.concurrent.locks.Lock)**:
   - Definition: Explicit locks are provided by the `Lock` interface and its implementations (e.g., `ReentrantLock`).
   - Usage:
     - You create an instance of a `Lock` (e.g., `ReentrantLock`) and explicitly acquire and release the lock.
     - Locks allow more flexibility and features than `synchronized` blocks.
   - Advantages:
     - Fine-grained control: You can lock and unlock explicitly, allowing more complex synchronization patterns.

        - Additional features: Locks provide methods like `tryLock()`, `lockInterruptibly()`, and fairness settings.
    - Considerations:
        - Use explicit locks when you need advanced features or more control.
        - Be cautious not to forget to release the lock (use `try`/`finally` blocks).

**Choosing Between Implicit and Explicit Locks**

- Use `synchronized`:
    - For simple synchronization needs.
    - When you want the JVM to optimize synchronized blocks.
    - When fairness is not a concern.
- Use `Lock`:
    - When you need advanced features (e.g., timeouts, fairness, multiple conditions).
    - For fine-grained control over locking.
    - When you want to avoid deadlocks or interruptible locks.

**Condition Locks**

Condition locks in Java are a synchronization mechanism built on top of regular locks (like `ReentrantLock`) that provide more fine-grained control over thread coordination. They allow threads to wait until a specific condition becomes true before acquiring a lock. Here's a breakdown of how they work:

Components:

- Lock: A Lock object (usually a `ReentrantLock`) manages exclusive access to shared resources.
- Condition: A Condition object associated with a Lock enables threads to wait based on specific conditions.

Functionality:

- Acquire Lock: A thread acquires the lock associated with the condition before checking the condition itself.
- Check Condition: The thread evaluates the condition. If it's not true:
    - Wait: The thread calls `await()` on the condition object to release the lock and wait for a signal from another thread. While waiting, the lock is released, allowing other threads to acquire it.
    - Spurious Wakeup: Threads can be woken up spuriously (without a specific signal), so it's crucial to re-evaluate the condition after `await()` returns.
- True Condition: If the condition becomes true, the thread continues execution.
- Release Lock: The thread always releases the lock using `unlock()` (within a `finally` block) before exiting the critical section.

Benefits:

- Conditional Waiting: Threads wait only if necessary, improving efficiency compared to waiting on the lock itself.
- Signaling: Threads can explicitly signal waiting threads when the condition becomes true, promoting better synchronization.

**Example**

The code example in Code 1 demonstrates the use of Java locks, specifically `ReentrantLock`, to manage a printing queue.

Key Points:

- Synchronization: The use of `lock()` and `unlock()` ensures only one thread can modify the print queue at a time.
- Waiting/Signaling: Threads that add jobs and the thread that prints use the lock's associated monitor to wait and signal respectively when the queue is empty. This mechanism prevents the printer thread from waiting indefinitely.

Write the code on your Java IDE, compile and run. Notice the differences among several runs.

## 1.2 Semaphores

Semaphores are a powerful tool for managing concurrent access, and they play a crucial role in multithreaded programming. A semaphore is a synchronization primitive that allows controlling access to a shared resource by limiting the number of concurrent threads that can access it. Unlike locks, which allow only one user per resource, semaphores can restrict any given number of users to a resource. For instance, if we have a parking lot with 6 parking spots, at most 6 parking permits can be issued for 6 cars simultaneously.

**Keynotes**

Unlike locks, which enforce exclusive access to a resource by a single thread, semaphores allow controlled concurrent access by a specified number of threads or processes. While semaphores can be used to emulate lock-like behavior, they are generally not ideal for this purpose. This is because they lack the concept of thread ownership, which can potentially lead to problems such as:

- Non-reentrancy: a thread can acquire the semaphore multiple times, leading to unexpected behavior.
- Any thread can release the semaphore, which can result in a race condition in the critical section.
- Semaphores do not provide a mechanism to ensure fairness in accessing the shared resource.

There's a concept of timed semaphores. These allow a number of permits within a given period of time, after which the time resets and all permits are released.

**Example**

The following code example demonstrates how to use a Java Semaphore to manage a parking lot simulation. The main ideas are:

1. Semaphore as Parking Spaces: The Semaphore object represents the available parking spaces. Its initial count is set to the total number of spaces in the parking lot.
2. Acquiring a Permit = Entering: When a car (thread) wants to enter, it attempts to acquire a permit from the Semaphore. If a permit is available, then the car "parks."
3. Releasing a Permit = Leaving: When a car leaves the parking lot, it releases a permit back to the Semaphore, signaling that a spot has opened up.

Write the code on your Java IDE, compile and run. Notice the differences among several runs.

# 2 Practice

1. Is there a way to find out how many processors are available to the Java Virtual Machine (JVM) for parallel processing? Explain.

2. Write a java multithreading program to simulate a producer-consumer scenario using a bounded buffer. The producer adds items to the buffer, and the consumer removes items from it. Use condition locks to ensure that the consumer waits when the buffer is empty and the producer waits when the buffer is full.

3. When using semaphores, what do we mean with fairness policy?

4. Assuming we have three printers, rewrite the code in Code 1 to manage a total of 12 printing jobs using Java semaphore. Make sure the system does not terminate until all of the 12 jobs are completed.

5. Write a program that demonstrates deadlock.

# 3 Code Examples

*Code 1: Lock Example*

```java
//Import necessary classes
import java.util.Date;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

//Class representing a printing job
class PrintingJob implements Runnable {
    private final PrinterQueue printerQueue; // Reference to the printer queue

    public PrintingJob(PrinterQueue printerQueue) {
        this.printerQueue = printerQueue; // Assign the printer queue
    }

    @Override
    public void run() {
        System.out.printf("%s: Going to print a document\n", Thread.currentThread().getName());
        printerQueue.printJob(new Object()); // Submit the printing job
    }
}

//Class representing the printer queue with thread-safe printing functionality
class PrinterQueue {
    private final Lock queueLock = new ReentrantLock(); // Lock for thread-safe access

    public void printJob(Object document) {
        queueLock.lock(); // Acquire lock for exclusive access
        try {
            Long duration = (long) (Math.random() * 10000); // Random printing duration
            System.out.println(Thread.currentThread().getName() + ": PrintQueue: Printing a Job during "
                    + (duration / 1000) + " seconds :: Time - " + new Date()); // Print job details
            Thread.sleep(duration); // Simulate printing time
         } catch (InterruptedException e) {
             e.printStackTrace(); // Handle potential interruption
         } finally {
             queueLock.unlock(); // Release lock to allow other threads access
         }
     }
}

//Main class demonstrating the program execution
public class PrintQueue {
    public static void main(String[] args) {
        // Create an instance of the printer queue
        PrinterQueue printerQueue = new PrinterQueue();

        // Create multiple printing jobs (threads)
        Thread job1 = new Thread(new PrintingJob(printerQueue), "Job 1");
        Thread job2 = new Thread(new PrintingJob(printerQueue), "Job 2");
        Thread job3 = new Thread(new PrintingJob(printerQueue), "Job 3");

        // Start the threads to initiate printing jobs
        job1.start();
        job2.start();
        job3.start();
    }
}
```

*Code 2: Semaphore Example*

```java
//Import necessary classes for concurrency and randomness
import java.util.concurrent.Semaphore;
import java.util.concurrent.TimeUnit;
import java.util.Random;

//Class representing the parking lot with a controlled number of spaces
public class ParkingLot {
// Semaphore to manage the available parking spaces
    private final Semaphore availableSpaces;

// Constructor initializing the Semaphore with the total number of spaces
    public ParkingLot(int totalSpaces) {
        // Create Semaphore with initial permit count
        availableSpaces = new Semaphore(totalSpaces);
    }

// Method for a car to attempt to enter the parking lot
    public boolean enter(int carId) {
        try {
            // Indicate a car is trying to park
            System.out.println("Car #" + carId + " trying to park.");
            // Attempt to acquire a permit for parking, waiting up to 2 seconds
            if (availableSpaces.tryAcquire(2, TimeUnit.SECONDS)) {
                // If a permit is acquired, indicate successful parking
                System.out.println("Car #" + carId + " parked.");
                // Simulate parking for a random duration
                parkForRandomTime();
                return true; // Signal successful parking
            } else {
                // If a permit isn't available within the timeout, indicate leaving
                System.out.println("Car #" + carId + " could not find a spot, leaving.");
                return false;
            }
        } catch (InterruptedException e) {
            // Handle potential interruption during waiting
            System.out.println("Car #" + carId + " was interrupted while waiting to park.");
            return false;
        }
    }

// Method for a car to leave the parking lot
    public void leave(int carId) {
        availableSpaces.release(); // Release a permit, making a space available
        System.out.println("Car #" + carId + " left the parking lot.");
    }

// Method to simulate parking for a random duration
    private void parkForRandomTime() {
        Random random = new Random();
        // Generate a random parking time between 1 and 5 seconds
        int parkingTime = random.nextInt(5) + 1;
        try {
            Thread.sleep(parkingTime * 1000); // Sleep for the simulated parking time
        } catch (InterruptedException e) {
            // Handle potential interruption during parking
            e.printStackTrace();
        }
    }
// Main method to initiate the parking lot simulation
    public static void main(String[] args) {
        ParkingLot parkingLot = new ParkingLot(5); // Create a parking lot with 5 spaces

        // Create and start 10 cars (threads) to simulate parking attempts
        for (int i = 1; i <= 10; i++) {
            new Car(i, parkingLot).start();
        }
    }
}

//Class representing a car (thread) in the parking lot simulation
class Car extends Thread {
    private final int carId;
    private final ParkingLot parkingLot;

    public Car(int carId, ParkingLot parkingLot) {
        this.carId = carId;
        this.parkingLot = parkingLot;
    }

// Thread's main logic: attempt to enter the parking lot, then leave
    @Override
    public void run() {
        if (parkingLot.enter(carId)) { // If parking is successful
            parkingLot.leave(carId); // Leave the parking lot
        }
    }
}
```