DEADLOCK

A **deadlock** occurs when two threads are trying to access resources that the other thread is currently using. For example:

We have **Thread 1** and **Thread 2**, as well as **Resource 1** and **Resource 2**.

- **Thread 1** is using **Resource 1**, so no one else can access it.
- **Thread 2** is using **Resource 2**, so no one else can access it.

So far, so good. The issue arises when **Thread 1** tries to access **Resource 2** without releasing **Resource 1**, while **Thread 2** tries to access **Resource 1** without releasing **Resource 2**.

This causes the program to become stuck because both threads are blocked, waiting for each other. Since each thread requires the other to release its resource, progress becomes impossible. Please refer to my code example for reference.

Note: This does not have to involve only two threads with two resources. Any number of threads and resources can cause this issue if a "circular dependency" occurs, where **Thread 1** blocks **Thread 2**, which in turn blocks **Thread 3**, and so on.

Analysing Thread Dumps

For the thread dump, I used **VisualVM**, which makes it much easier to view all active threads.

When analysing a thread dump, look for the following keywords:

1. **Blocked**: Indicates that the thread itself is blocked.
2. **Locked**: Shows the resource ID that the thread has locked.
3. **Waiting to lock**: Indicates the resource that the thread is unable to access because it is locked by another thread.

How to Avoid Deadlocks

1. **Minimize Lock Scope**:
   Keep synchronized blocks (sections of code where the resource is accessible only to the current thread) as short as possible.
2. **Use Semaphores**:
   Semaphores act as an access permit pool for threads, allowing a certain number of threads to access a specific code block simultaneously.
3. **Use Locks**:
   Locks allow a thread to access a code block only when the lock is available (unlocked).
4. **Avoid Circular Waiting**:
   Ensure that threads acquire locks in the same order to prevent circular dependencies.

For examples of semaphores and locks, please consult the provided code.

Memory Leaks in Java

A **memory leak** occurs when objects that are no longer needed remain referenced, preventing the JVM from reclaiming their memory. This leads to excessive memory usage while the program is running. Thankfully, when the program exits, the JVM releases all memory back to the operating system.

How I Identified Memory Leaks

I used **VisualVM** to monitor memory usage and analyse heap dumps. To detect a memory leak, I looked for the following indicators:

1. **High memory usage** that doesn't decrease after garbage collection.
2. **High instance counts** for specific classes.
3. **Unreachable but retained objects**, where reference chains show why the objects remain in memory.

How Memory Leaks Occur

Memory leaks in Java often result from:

1. **Unreleased Resources**: Failing to release objects like lists, buffers, or connections.
2. **Listeners/Callbacks**: Not unsubscribing or unregistering them when no longer needed.
3. **Static References**: Objects held in static fields or collections that are not cleared.

How to Avoid Memory Leaks

1. **Release Resources Properly**: Ensure all used resources are explicitly cleared or closed when no longer needed.
2. **Unsubscribe Listeners**: Remove any listeners, observers, or callbacks after their purpose is fulfilled.
3. **Monitor During Development**: Regularly test memory usage and analyse heap dumps to catch potential leaks early.

Impact of Memory Leaks

Memory leaks lead to **performance degradation**, with increased garbage collection cycles, and may eventually cause an **OutOfMemoryError**, crashing the application. Monitoring memory usage during development and testing is key to preventing such issues.

High CPU Usage

High CPU usage occurs when a program uses an excessive amount of the CPU’s processing power. This can lead to performance issues, such as system slowdowns, unresponsiveness, or overheating on physical machines. Identifying and addressing high CPU usage is crucial to ensure smooth application performance and efficient resource utilization.

Monitoring CPU Usage

1. **VisualVM**:
    - VisualVM can show the CPU usage of the running Java program in real-time.
2. **Windows Task Manager**:
    - **Processes Tab**: View the CPU consumption of the specific program.
    - **Performance Tab**: Monitor overall CPU usage on the system.

Causes of High CPU Usage

High CPU usage in Java programs can be caused by:

1. **Loops with Intense Calculations**: Performing heavy mathematical operations in loops.
2. **Infinite Loops**: Loops without exit conditions that continuously consume CPU cycles.
3. **Recursive Functions**: Functions with no or improper base conditions, leading to excessive stack calls.
4. **Handling Large Data Sets**: Processing or sorting large data structures inefficiently.

Monitoring tools and careful code optimization are essential to detect and resolve these issues effectively.