JAVA VIRTUAL MACHINE (JVM) INTERNALS AND GARBAGE COLLECTOR BEHAVIOR

Sandra Kumi

REVIEWER Mr. Thomas Darko

JAVA VIRTUAL MACHINE (JVM) INTERNALS AND GARBAGE COLLECTOR BEHAVIOR

1. JVM Architecture Overview

The JVM is the runtime environment that enables Java bytecode to be executed on different platforms. Its main components include:

- **Class Loader**: Loads class files and places them into memory.
- **Execution Engine**: Interprets bytecode or compiles it into native code using the Just-In-Time (JIT) compiler.
- **Memory Areas**:
    - **Heap**: Stores objects and class instances.
    - **Stack**: Holds local variables and method call information.
    - **Metaspace**: Holds metadata about classes and methods.
    - **Program Counter Register**: Tracks the next instruction for execution.
    - **Native Method Stack**: Supports native methods execution.

2. Garbage Collection in the JVM

Garbage Collection (GC) is a mechanism to automatically reclaim memory by removing objects that are no longer in use. The JVM uses generational GC to optimize performance by dividing memory into two regions:

- **Young Generation**: Newly created objects are first allocated here. It's further divided into Eden and Survivor spaces.
- **Old Generation**: Objects that survive multiple GC cycles in the young generation are moved here.

The JVM triggers garbage collection when the heap becomes full, employing various garbage collection algorithms.

3. Types of Garbage Collectors

- **Serial GC**: A single-threaded GC suitable for small applications.
- **Parallel GC**: Uses multiple threads to perform GC tasks, improving throughput for larger applications.
- **G1 GC (Garbage First)**: Focuses on minimizing pause times and is optimal for applications that require a balance between throughput and low-latency.
- **ZGC (Z Garbage Collector)** and **Shenandoah**: Both are designed for very low pause times and are highly scalable for large heaps.

Comparison of Garbage Collector Performance

Test Setup

- **Environment**: A Java application with high memory usage (e.g., a web server handling multiple concurrent users).
- **GC Options**: Tested the following collectors:
    - Serial GC
    - Parallel GC
    - G1 GC
    - ZGC

Metrics Compared

- **GC Pause Times**: The time spent in GC cycles. Lower pause times are preferred for responsive applications.
- **Throughput**: The percentage of time the application spends executing (not collecting garbage).
- **Heap Utilization**: The amount of heap memory used after a GC cycle.
- **Latency**: Time taken by each GC cycle to reclaim memory.

Results

1. **Serial GC**:

- **Pause Time**: High, as it uses a single thread.
- **Throughput**: Moderate, best suited for single-threaded applications.

2. **Parallel GC**:
   - **Pause Time**: Lower compared to Serial GC due to multi-threading.
   - **Throughput**: High, but may cause longer pause times in exchange for higher throughput.
3. **G1 GC**:
   - **Pause Time**: Balanced, designed to maintain short pause times.
   - **Throughput**: Good, especially for applications with mixed latency and throughput needs.
4. **ZGC**:
   - **Pause Time**: Extremely low, ideal for low-latency applications.
   - **Throughput**: Moderate, but excellent for large heap sizes.

Memory Management Best Practices

1. Heap Sizing

Set appropriate initial (-Xms) and maximum (-Xmx) heap sizes to ensure the JVM doesn't spend unnecessary time resizing the heap. Too small a heap can lead to frequent GC, while too large can result in long pause times.

2. Garbage Collection Tuning

- **Maximize Throughput**: For applications prioritizing high throughput (e.g., batch processing), choose Parallel GC and tune parameters like -XX:GCTimeRatio to balance execution and GC time.
- **Minimize Latency**: For applications requiring low-latency (e.g., interactive applications), use G1 GC or ZGC with parameters like -XX:MaxGCPauseMillis to control pause time.

3. Memory Efficient Coding

- **Avoid Unnecessary Object Creation**: Use primitive types when possible and avoid creating unnecessary objects in loops.
- **Use Object Pools**: Reuse expensive-to-create objects to reduce GC overhead (e.g., database connections).
- **Optimize Data Structures**: Use memory-efficient collections (e.g., ArrayList vs. LinkedList) and avoid excessive memory retention by clearing unused elements.

4. Profiling Tools

- **VisualVM**: Track heap usage, object allocation, and garbage collection events.
- **JProfiler**: Analyze memory leaks, object creation rates, and optimize memory usage.
- **Eclipse MAT**: Analyze heap dumps to find memory leaks and inefficient object usage patterns.