Major Reasons for JVM Minor GC (Garbage Collection)

Minor GC (Young Generation GC) occurs when the **Eden space** in the Young Generation of the heap is **filled up**, prompting a collection to reclaim memory. Here are the key technical reasons for Minor GC:

---

1. High Object Creation Rate

- Frequent object allocations lead to **rapid filling of the Eden space**.
- When Eden is full, a Minor GC is triggered to move surviving objects to **Survivor spaces (S0/S1)** or **Tenured/Old Generation**.

Example:

```
for (int i = 0; i < 100000; i++) {
   String str = new String("PerformanceTest");  // Rapid object creation
}
```

🔹 **Issue:** High allocation rate increases GC frequency.
🔹 **Resolution:** Use object pooling or reuse objects (e.g., StringBuilder instead of String concatenation).

---

2. Insufficient Survivor Space Size

- When objects **survive multiple Minor GCs**, they are moved from **Eden → Survivor (S0/S1) → Old Generation**.
- If Survivor space is too **small**, objects will be prematurely promoted to the Old Generation.
- This increases the risk of **Premature Old Gen Promotion**, leading to **frequent Major (Full) GCs**.

Example JVM Configuration Problem:

```
-XX:SurvivorRatio=8  # (Too high, makes survivor spaces too small)
```

🔹 **Issue:** Insufficient Survivor space → objects get prematurely promoted to Old Gen.
🔹 **Resolution:** Adjust -XX:SurvivorRatio or increase heap size.

---

3. Long-lived Objects in Young Generation

- Some objects persist for multiple GC cycles but still remain in Young Gen.

- JVM **tracks the object age** using an **age threshold (XX:MaxTenuringThreshold)**.
- If objects **stay longer than the threshold**, they are promoted to Old Gen.

Example JVM Setting:

```
-XX:MaxTenuringThreshold=5  # Objects promoted after 5 GC cycles
```

- **Issue:** High tenuring threshold increases Minor GC overhead.
- **Resolution:** Reduce MaxTenuringThreshold if unnecessary promotions occur.

---

4. Large Arrays and Large Object Allocations

- Large object allocations (e.g., large arrays or buffers) **directly impact Eden space**.
- If **too large**, they may be allocated directly in **Old Gen (TLAB bypassing)**.

Example:

```
int[] largeArray = new int[10000000];  // Large object allocation
```

- **Issue:** Direct promotion to Old Gen increases risk of Full GC.
- **Resolution:** Use object pooling or optimize array allocation.

---

5. Thread-local Allocation Buffers (TLABs) Filling Up

- JVM **allocates memory per-thread** in **Thread-local Allocation Buffers (TLABs)**.
- If TLABs are **small or fragmented**, Eden gets filled faster → triggering Minor GC.

Example JVM Setting:

```
-XX:+UseTLAB -XX:TLABSize=512K
```

- **Issue:** TLAB exhaustion increases Minor GC frequency.
- **Resolution:** Tune TLABSize or reduce excessive thread creation.

---

6. Poor GC Algorithm Selection

- Using **inappropriate GC algorithms** may cause **inefficient Minor GC behavior**.
- For example:
  - **Serial GC** (default in small heaps) → Single-threaded, slow.
  - **Parallel GC** (better for multi-core) → Faster Minor GC.
  - **G1 GC** (adaptive) → Balanced performance.

Example JVM Setting:

```
-XX:+UseParallelGC  # Parallel GC for better Minor GC handling
```

- **Issue:** Wrong GC algorithm can lead to excessive Young Gen pauses.
- **Resolution:** Use -XX:+UseG1GC for balanced GC behavior.

---

7. Improper Heap Size Configuration

- **Too small heap** → Frequent Minor GC.
- **Too large heap** → Increased GC pause times.

Example JVM Setting:

```
-Xms512m -Xmx2g -XX:NewRatio=2
```

- **Issue:** Misconfigured heap leads to either high GC frequency or long GC pauses.
- **Resolution:** Tune -Xms, -Xmx, and -XX:NewRatio based on application needs.

---

8. High Allocation Rate in Multi-threaded Applications

- Multi-threaded apps create **more objects per second**, filling Eden faster.
- **Thread contention** may also cause GC inefficiencies.

Example:

```
ExecutorService executor = Executors.newFixedThreadPool(100);
```

- **Issue:** Too many threads increasing allocation rate.
- **Resolution:** Use **pooled resources** and limit unnecessary object creation.

Conclusion

Reason	Impact on Minor GC	Solution
High Object Creation Rate	Frequent Minor GCs	Optimize object allocation
Small Survivor Space	Premature promotion to Old Gen	Adjust -XX:SurvivorRatio
Long-lived Young Gen Objects	Increased GC overhead	Optimize -XX:MaxTenuringThreshold
Large Objects Allocation	May cause Full GC	Pool large objects

Reason	Impact on Minor GC	Solution
TLAB Exhaustion	More Minor GC	Tune TLABSize
Wrong GC Algorithm	Suboptimal GC performance	Use G1GC or ParallelGC
Improper Heap Size	High GC frequency or long pauses	Optimize -Xms, -Xmx
Multi-threaded Allocation Overhead	Increased allocation rate	Limit excessive object creation

JVM GC Log Analysis for Minor GC Events

Step 1: Enable GC Logging
To analyze Minor GC, first, enable JVM GC logs in your application:

```
# For Java 8

-XX:+PrintGCDetails -XX:+PrintGCDateStamps -Xloggc:gc.log


# For Java 11+

-Xlog:gc*:file=gc.log:time,uptime,level,tags
```

Step 2: Understanding GC Log Entries for Minor GC

Example GC Log Entry (Minor GC)

```
2025-02-10T12:30:15.234+0000: 5.678: [GC (Allocation Failure)

[PSYoungGen: 102400K->16384K(153600K)] 250000K->180000K(512000K), 0.015s]

[Times: user=0.02 sys=0.00, real=0.01 secs]
```

Step 3: Breaking Down the Log Entry

Field	Meaning
2025-02-10T12:30:15.234+0000	Timestamp of GC event
5.678:	Time since JVM start (in seconds) when GC occurred
[GC (Allocation Failure)	GC reason (Eden filled up)
[PSYoungGen: 102400K->16384K(153600K)]	Eden before → after GC (Young Gen size before and after GC)
250000K->180000K(512000K)	Total heap before → after GC (Including Old Gen)
0.015s	GC duration (15ms)
[Times: user=0.02 sys=0.00, real=0.01 secs]	CPU time breakdown

---

Step 4: Common Minor GC Patterns and Issues

1. Frequent Minor GC (Too Many GCs)

Log Pattern:

```
2025-02-10T12:31:00.678+0000: 10.567: [GC (Allocation Failure) [PSYoungGen: 40960K->8192K(51200K)] 120000K->95000K(256000K), 0.020s]

2025-02-10T12:31:02.001+0000: 12.012: [GC (Allocation Failure) [PSYoungGen: 45000K->10240K(51200K)] 140000K->110000K(256000K), 0.018s]
```

Issue:

- Too many **small Minor GCs** within seconds
- Eden fills up too quickly → GC runs frequently

Resolution:

✅ **Increase Young Gen size** (-Xmn)
✅ Tune -XX:NewRatio to allocate more space to Young Gen
✅ Reduce excessive object creation

Example JVM tuning:

```
-XX:NewRatio=2 -Xmn512m
```

---

2. Survivor Space Overflows (Premature Promotion to Old Gen)

Log Pattern:

```
[GC (Allocation Failure) [PSYoungGen: 102400K->16384K(153600K)]

250000K->180000K(512000K), 0.015s]

[Times: user=0.02 sys=0.00, real=0.01 secs]

[GC (Allocation Failure) [PSYoungGen: 102400K->16400K(153600K)]

250000K->195000K(512000K), 0.016s]
```

- Old Gen size is **increasing after every Minor GC** → meaning objects are being **promoted too early**
- This can **trigger Full GC** sooner

Resolution:

✅ Increase -XX:SurvivorRatio to allow more objects to stay in Survivor
✅ Increase -XX:MaxTenuringThreshold to prevent premature promotion

Example JVM tuning:

```
-XX:SurvivorRatio=6 -XX:MaxTenuringThreshold=10
```

---

3. Large Objects Causing Young Gen Overflow

Log Pattern:

```
[GC (Allocation Failure) [PSYoungGen: 81920K->40960K(102400K)]

300000K->280000K(512000K), 0.040s]
```

- **Young Gen is not shrinking much after GC**
- Large objects **remain in Young Gen**

Resolution:

✅ Use **TLAB tuning** (-XX:+UseTLAB -XX:TLABSize=512K)
✅ Allocate large objects **directly to Old Gen**:

```
-XX:PretenureSizeThreshold=1M
```

✅ Use **Object Pooling** instead of large object creation

---

4. TLAB Exhaustion (Thread-Local Buffer Overflow)

Log Pattern:

```
[GC (Allocation Failure) [PSYoungGen: 10240K->4096K(20480K)] 50000K->45000K(256000K),
0.025s]
```

- **Minor GC occurs with small Young Gen utilization**
- Suggests **TLAB filling up too quickly**

Resolution:

✅ Increase **TLAB size**

```
-XX:TLABSize=1M
```

✅ Reduce excessive thread creation

Step 5: Analyzing GC Log Using GC Tools

1. Use GCEasy Online Analyzer

- Upload gc.log to GCEasy.io for analysis

2. Use GCViewer

- Download **GCViewer** tool:
    - Parses GC logs and shows **heap trends, GC pause times, and object promotion trends**

3. Use JVisualVM to Monitor GC in Real Time

```
jvisualvm
```

- Go to **Profiler > GC** to see live Minor GC trends

Step 6: JVM Optimization for Efficient Minor GC

Issue	GC Log Pattern	Resolution
Frequent Minor GC	GC every few seconds	Increase -Xmn, tune -XX:NewRatio
Survivor Space Overflow	High promotion to Old Gen	Adjust -XX:SurvivorRatio, -XX:MaxTenuringThreshold
Large Object Overflows	High Young Gen retention	Tune -XX:PretenureSizeThreshold, use object pooling
TLAB Filling Quickly	GC even with low Young Gen usage	Increase -XX:TLABSize, optimize thread creation

---

Final JVM Tuning Example for Better Minor GC Performance

```
# Increase Young Gen
-Xmn1g -XX:NewRatio=2


# Tune Survivor Space
-XX:SurvivorRatio=6 -XX:MaxTenuringThreshold=10


# Optimize Large Object Handling
-XX:PretenureSizeThreshold=1M
```

```
# TLAB Optimization

-XX:+UseTLAB -XX:TLABSize=512K


# Enable GC Logging (Java 8)

-XX:+PrintGCDetails -XX:+PrintGCDateStamps -Xloggc:gc.log
```

Conclusion

- **Enable GC Logs** (-Xloggc:gc.log) and analyze patterns.
- **Identify Frequent Minor GC** issues using **heap trends**.
- **Tune Survivor Space & Tenuring Threshold** to prevent premature promotions.
- **Use tools like GCViewer, GCEasy, and JVisualVM** for deeper analysis.