GC Thrashing and Memory Pressure in a JVM

How to detect **GC Thrashing** and **Memory Pressure** in a JVM, covering:

- 🔍 GC internals
- 🔬 Allocation & promotion diagnostics
- 📉 Impact on CPU, latency, and throughput
- 🧠 Deep jstat interpretation
- 🧵 Thread dump correlation
- 📁 MAT + GC log correlation

🔥 1. What is GC Thrashing — *Technically*

GC thrashing happens when the JVM **cannot keep up** with the allocation rate, leading to:

- 🧠 **Back-to-back garbage collections** (Young or Full)
- 📦 **Old Generation fills up rapidly**, but GC fails to reclaim enough space
- 🔁 **High GC frequency and duration** with little pause between events
- 🐢 Application threads spend most of their time **waiting at GC safepoints**

Root Cause Mechanics:

- High **object allocation rate** (e.g., new objects per second)
- High **object promotion rate** due to premature tenuring
- **Survivor spaces too small** → forced promotions to Old Gen
- **Fragmentation** in Old Gen → insufficient contiguous memory → Full GC
- GC unable to compact or reclaim → JVM ends up in a near-continuous GC cycle

---

🧠 2. Memory Pressure — *Deep Dive*

Memory pressure arises when:

- The **allocated heap** (-Xmx) is **not enough** for current workload

- **Objects live longer** than expected → Tenured fills up fast
- GC cannot free up enough space → Allocation failure
- **Safepoint stalls** block app threads during GC coordination

Indicators:

Symptom	Technical Description
FGC every 5–10 seconds	Old Gen is exhausted, not reclaimable
YGC with high Eden %	Eden fills rapidly – high allocation rate
Promotion Failure	Survivors or Old Gen can't accommodate
CMS concurrent mode failure	CMS couldn't finish GC before tenured filled up
G1 to-space exhausted	G1 couldn't reserve contiguous regions
Evacuation failure	G1GC failed during object copying

🧰 3. GC Log Analysis — *Advanced Interpretation*

Sample Log (G1GC):

```
2024-04-19T19:03:23.456+0000: 845.123: [GC pause (G1 Evacuation Pause) (young), 0.125s]
  [Parallel Time: 101.0 ms, GC Workers: 8]
  [GC Worker Start (ms): Min: 845123.4, Avg: 845123.6, Max: 845123.8, Diff: 0.4]
  [Eden: 384.0M(384.0M)->0.0B(320.0M) Survivors: 40.0M->48.0M Heap: 1024.0M(2048.0M)->752.0M(2048.0M)]
```

What to Notice:

- **Frequent young GC** → Eden fills fast (384 MB to 0 MB)
- Survivor space is small (40M → 48M) → Possible **promotion pressure**
- Heap occupancy stays high → **Old Gen not shrinking**

📈 4. jstat Analysis — Real-Time Deep Inspection

Run:

```
jstat -gcutil <pid> 1s 100
```

Sample output:

```
 S0    S1    E     O     M    CCS   YGC   YGCT   FGC   FGCT    GCT
 5.00  60.00  85.00  92.00  97.00  95.00  309   12.34   85   45.56   57.90
```

Advanced Readings:

Metric	Diagnosis
E > 80% continuously	Eden is too small or allocation rate is high
O > 85% steadily	Old Gen saturation → Full GC trigger
YGC increases rapidly	High churn, frequent Young GCs
FGC count increases rapidly	GC thrashing; can't reclaim OldGen
FGCT vs GCT → >80% of time	JVM stuck in GC

Pattern to confirm GC thrashing:

- FGC increasing every few seconds
- O column stays high
- GCT climbing steeply

🪓 5. Heap Dump + GC Log Correlation

1. Generate dump during thrashing:

```
jmap -dump:live,format=b,file=thrashing-heap.hprof <pid>
```

2. Use **Eclipse MAT**:

- Load HPROF
- Open *Dominator Tree*
- Sort by *Retained Heap*
- Look for:

  - HashMap, ThreadLocal, ArrayList with high retained size
  - Static references holding memory
  - Duplicate classloaders (in redeploy scenarios)

Example:

Class: com.example.cache.CustomCache → Retained Heap: 512MB

🚩 Red flag: cache is not releasing memory → memory pressure

Now correlate the timestamp in GC logs with heap dump snapshot.

📊 6. APM and JMX Deep Monitoring Strategy

Metrics to Alert On:

Metric	Threshold
Heap Used (%)	>85% consistently
GC Time (%)	>60–70%
FGC Count	>3 in 5 min
GC Pause Time	>1s frequently
Allocation Rate	>100MB/s
Promotion Rate	>50MB/s
Tenured Generation Utilization	>80% consistently
Survivor Overflow	Frequent
Eden Reclaim %	Low (<30%)

Tools:

- **Prometheus + Grafana Dashboards**
- **Dynatrace memory hotspots**
- **New Relic GC metrics**
- **JConsole + VisualVM**

🧵 7. Thread Dump Correlation

Run:

```
jstack -l <pid> > tdump.log
```

Check:

- **Application threads blocked on GC safepoint**:
- "http-nio-8080-exec-2" #34 waiting on condition
- ↳ sun.misc.Unsafe.park()
- ↳ GCLocker::stall

This means GC is freezing threads too often → throughput loss.

- GC threads are **highly active**:
- "GC Thread#2" daemon prio=10 tid=0x00007f89dc01b800 runnable

If GC threads dominate CPU and app threads are mostly waiting → GC thrashing confirmed.

🧪 8. Real-Time JVM Allocation Profiling (Bonus)

Use **Java Flight Recorder (JFR)** or **VisualVM** with allocation sampling:

- Profile which classes are allocating most objects
- Check:
  - byte[] → buffer/caching issues
  - String → unbounded request data
  - List → holding references too long

🧠 9. JVM GC Tuning Strategy (if thrashing detected)

GC	**Fix**
CMS	Tune -XX:CMSInitiatingOccupancyFraction=40

G1	Tune -XX:InitiatingHeapOccupancyPercent=30
G1	Use -XX:MaxGCPauseMillis=200 to trigger earlier
G1	Avoid -XX:+UseLargePages unless memory pinned
General	Increase -Xms and -Xmx to avoid heap resizing

📌 10. Summary: How to Detect and Confirm GC Thrashing

Layer	Check
GC Logs	Frequent full GCs, low memory reclaimed
jstat	FGC/FGCT and O% trends
APM	High GC % time, pause durations
Heap Dump	Memory leaks or high-retention objects
Thread Dump	Threads waiting on GC, GC threads dominant
JFR	Allocation profiling shows high churn
Logs	java.lang.OutOfMemoryError, GC overhead limit exceeded