Advanced Thread Dump Analysis: Identifying and Resolving Multiple Performance Bottlenecks in Java Applications

Scenario: Complex Thread Dump Analysis with Multiple Issues

Background

A large-scale Java Spring Boot application running on AWS EC2 instances has been experiencing slow response times and occasional timeouts. The system handles high concurrent traffic and utilizes a multi-threaded architecture for processing requests.

Observed Issues

- **Frequent CPU spikes**
- **Increased response times for API requests**
- **Occasional unresponsiveness requiring manual restarts**
- **High thread count observed in application monitoring**
- **Increase in error rates related to database timeouts**

Step 1: Collect the Thread Dump

Since no external tools are allowed, we will use the jstack command to capture the thread dump from the running JVM process.

```
jstack -l <PID> > thread_dump.txt
```

Alternatively, if running on a Windows system:

```
jcmd <PID> Thread.print > thread_dump.txt
```

On Kubernetes:

```
kubectl exec -it <pod-name> -- jstack -l 1 > /tmp/thread_dump.txt
```

Ensure you capture multiple thread dumps at different time intervals (e.g., every 5-10 seconds) to observe patterns.

Step 2: Identify and Categorize Issues in the Thread Dump

We now analyze the collected thread dump, focusing on five common thread-related issues.

Issue 1: High Number of BLOCKED Threads (Thread Contention/Deadlocks)

Look for BLOCKED threads in the thread dump:

```
"Thread-53" #67 prio=5 os_prio=0 tid=0x00007f9c4c01e000 nid=0x4e67 waiting for monitor entry [0x00007f9b3cfe6000]

   java.lang.Thread.State: BLOCKED (on object monitor)
```

at com.example.service.PaymentService.processPayment(PaymentService.java:120)

- waiting to lock <0x000000076ab64b90> (a com.example.database.DBConnection)

- locked <0x000000076ab64c10> (a com.example.database.DBTransaction)

- **Analysis:** This suggests that PaymentService.processPayment() is waiting for a lock on DBConnection, indicating a possible **thread contention issue**.
- **Resolution:**
  - Check synchronized code blocks in PaymentService.java:120
  - Reduce lock contention by using concurrent data structures (ConcurrentHashMap, ReadWriteLock)
  - Optimize lock granularity (use fine-grained locks instead of a single global lock)

Issue 2: Deadlocks in Threads

Search for cycles in BLOCKED and WAITING threads:

"Thread-10" #10 prio=5 os_prio=0 tid=0x00007f9c4c00e000 nid=0x3e23 waiting for monitor entry [0x00007f9b3cfd5000]

java.lang.Thread.State: BLOCKED (on object monitor)

at com.example.service.UserService.updateUser(UserService.java:45)

- waiting to lock <0x000000076ab64c10> (a com.example.database.DBTransaction)

- locked <0x000000076ab64b90> (a com.example.database.DBConnection)

"Thread-11" #11 prio=5 os_prio=0 tid=0x00007f9c4c00f000 nid=0x3e24 waiting for monitor entry [0x00007f9b3cfd6000]

java.lang.Thread.State: BLOCKED (on object monitor)

at com.example.service.PaymentService.processPayment(PaymentService.java:120)

- waiting to lock <0x000000076ab64b90> (a com.example.database.DBConnection)

- locked <0x000000076ab64c10> (a com.example.database.DBTransaction)

- **Analysis:** Thread-10 is waiting for DBTransaction, held by Thread-11, while Thread-11 is waiting for DBConnection, held by Thread-10. **This is a deadlock.**
- **Resolution:**
  - Ensure that locks are acquired in a consistent order (e.g., always acquire DBConnection first before DBTransaction)
  - Use try-lock mechanisms (ReentrantLock.tryLock()) to prevent indefinite blocking

- Refactor database access patterns

Issue 3: Excessive Runnable Threads (CPU Spikes)

Look for too many threads in RUNNABLE state:

"Thread-99" #99 prio=5 os_prio=0 tid=0x00007f9c4c050000 nid=0x5e91 runnable [0x00007f9b3d012000]

java.lang.Thread.State: RUNNABLE

at java.util.HashMap.get(HashMap.java:600)

at com.example.service.CacheService.fetchData(CacheService.java:78)

at com.example.controller.DataController.getData(DataController.java:35)

- **Analysis:**
  - CacheService.fetchData() is using a HashMap.get(), which suggests a potential unbounded loop or a large number of active requests.
  - Possible cause: high CPU due to excessive object creation or inefficient loops.
- **Resolution:**
  - Optimize data access patterns (ConcurrentHashMap, caching strategies)
  - Profile CPU-intensive methods (VisualVM, perf for Linux)
  - Implement rate limiting and request throttling

Issue 4: Too Many WAITING Threads (Thread Pool Exhaustion)

Check for excessive WAITING threads:

"Thread-150" #150 prio=5 os_prio=0 tid=0x00007f9c4c00c000 nid=0x4e55 waiting on condition [0x00007f9b3cfcf000]

java.lang.Thread.State: WAITING (parking)

at sun.misc.Unsafe.park(Native Method)

at java.util.concurrent.locks.LockSupport.park(LockSupport.java:175)

at java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObject.await(AbstractQueuedSynchronizer.java:2039)

at java.util.concurrent.ArrayBlockingQueue.take(ArrayBlockingQueue.java:403)

at com.example.threadpool.CustomThreadPool.getTask(CustomThreadPool.java:98)

- **Analysis:**
    - CustomThreadPool.getTask() is blocked on ArrayBlockingQueue.take(), indicating **thread pool exhaustion**.
    - Possible causes: improper thread pool configuration, blocking calls inside tasks.
- **Resolution:**
    - Tune thread pool settings (corePoolSize, maxPoolSize)
    - Replace blocking queues with non-blocking alternatives (LinkedTransferQueue)
    - Identify and remove long-running tasks

Issue 5: Stuck Threads (Long GC Pause or Database Query Lag)

Identify threads stuck in database queries:

"Thread-200" #200 prio=5 os_prio=0 tid=0x00007f9c4c060000 nid=0x6e12 runnable [0x00007f9b3d020000]

Java.lang.Thread.State: RUNNABLE

At oracle.jdbc.driver.t4cpreparedstatement.execute(t4cpreparedstatement.java:1234)

At com.example.repository.orderrepository.fetchorders(orderrepository.java:150)

- **Analysis:**
    - T4CPreparedStatement.execute() suggests a long-running query.
    - Possible cause: **missing database indexes, high query complexity, blocking operations.**
- **Resolution:**
    - Check query execution plan (EXPLAIN ANALYZE)
    - Optimize indexes and database partitions
    - Set query timeout limits (setQueryTimeout in JDBC)

Step 3: Summary of Resolutions

Issue	Resolution
Blocked Threads (Thread Contention)	Reduce synchronized blocks, use ReadWriteLock

Deadlocks	Acquire locks in consistent order, use ReentrantLock.tryLock()
Excessive Runnable Threads (CPU Spikes)	Optimize loops, caching, use CPU profiling
Thread Pool Exhaustion (Waiting Threads)	Tune thread pool settings, remove blocking calls
Stuck Threads (Database Queries)	Optimize queries, indexes, and connection pool

By following these steps systematically, you can analyze and resolve performance bottlenecks using just the thread dump. 🚀