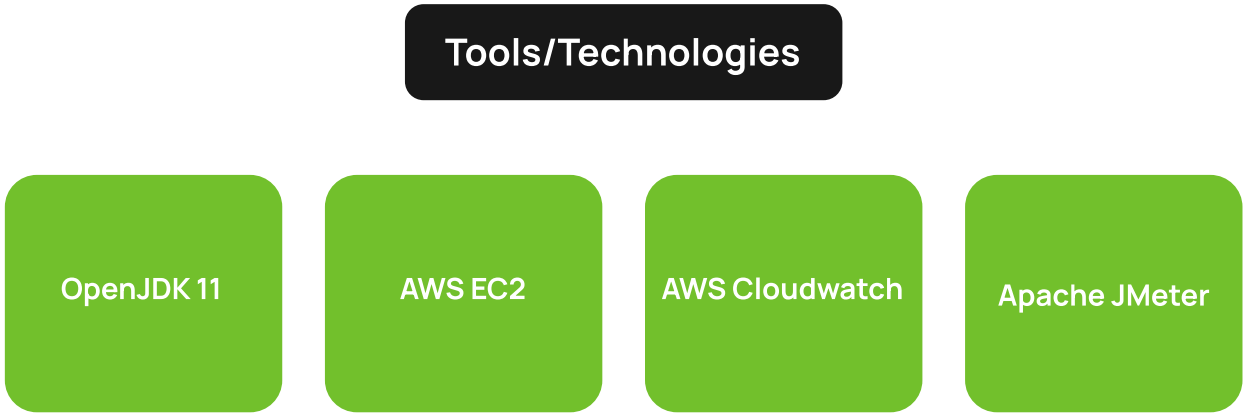
13. Overhead added by collecting thread dumps

A thread dump is a snapshot of all the threads running in a java process. It's a vital artifact to troubleshoot various production problems such as CPU spikes, unresponsiveness in the application, poor response time, hung threads, high memory consumption. Thus to facilitate troubleshooting, we have seen enterprises capture thread dumps on a periodic basis (every 5 minute or 2 minute). So we were curious to learn the overhead of capturing thread dump on a periodic basis. Thus we set out to conduct the below case study.

Environment

For our study we chose to use the open source spring boot pet clinic application. Pet Clinic is a poster child application that was developed to demonstrate the spring boot framework features.

We ran this application in OpenJDK 11. We deployed this application on the Amazon AWS t2.medium EC2 instance which has 16GB RAM and 2 CPUs. Test was orchestrated using Apache JMeter stress testing tool. We used AWS Cloudwatch to measure the CPU, Memory utilization. In nutshell here are the tools/technologies, we used to conduct this case study:



Test Scenario

In this environment, we conducted 3 tests:

1. **Baseline Test –** In this scenario we ran the pet clinic application without capture thread dumps using the JMeter tool for 20 minutes with 200 concurrent users.

2. **Thread dumps every 5 minutes Test –** In this scenario we ran the pet clinic application using the same JMeter script for 20 minutes with 200 concurrent users. However we captured thread dump from a pet clinic application every 5 minutes.

3. **Thread dumps every 2 minutes Test –** In this scenario we ran the pet clinic application using the same JMeter script for 20 minutes with 200 concurrent users. However we captured thread dump from a pet clinic application every 2 minutes.

Note: If you don't know how to capture thread dump, see **How to capture thread dumps? 8 options** for more details.

Test Results

We captured average CPU and memory utilization from the AWS Cloudwatch and average response time and throughput from the JMeter tool. Data collected from all the test scenarios are summarized in the below table.

Data collected	Baseline test	Every 5 minutes test	Every 2 minutes test
Avg CPU Usage	8.35%	10.40%	7.92%
Avg Memory Usage	20.80%	19.90%	19.60%
Avg Response Time	3901 ms	3888 ms	3770 ms
Avg Throughput	24.4/sec	25.8/sec	24.8/sec

As you can see there is no noticeable difference in the CPU and Memory consumption. Similarly there is no noticeable difference in the average response and transaction throughput.

Conclusion

Thus based on our study we can conclude that there is no noticeable overhead in capturing thread dumps on a 5 minutes or 2 minutes interval.

14. Java Virtual Threads – quick introduction

Java virtual threads is a new feature introduced in JDK 19. It has potential to improve an applications availability, throughput and code quality on top of reducing memory consumption. This post intends to introduce java virtual threads in an easily understandable manner.

Watch video

https://www.youtube.com/watch?v=1RHgvf8GKe4&t=1s

Thread Life Cycle

Let's walkthrough a typical lifecycle of a thread:

1. Thread is created in a thread pool

2. Thread waits in the pool for a new request to come

3. Once the new request comes, the thread picks up the request and it makes a backend

Database call to service this request.

4. Thread waits for response from the backend Database

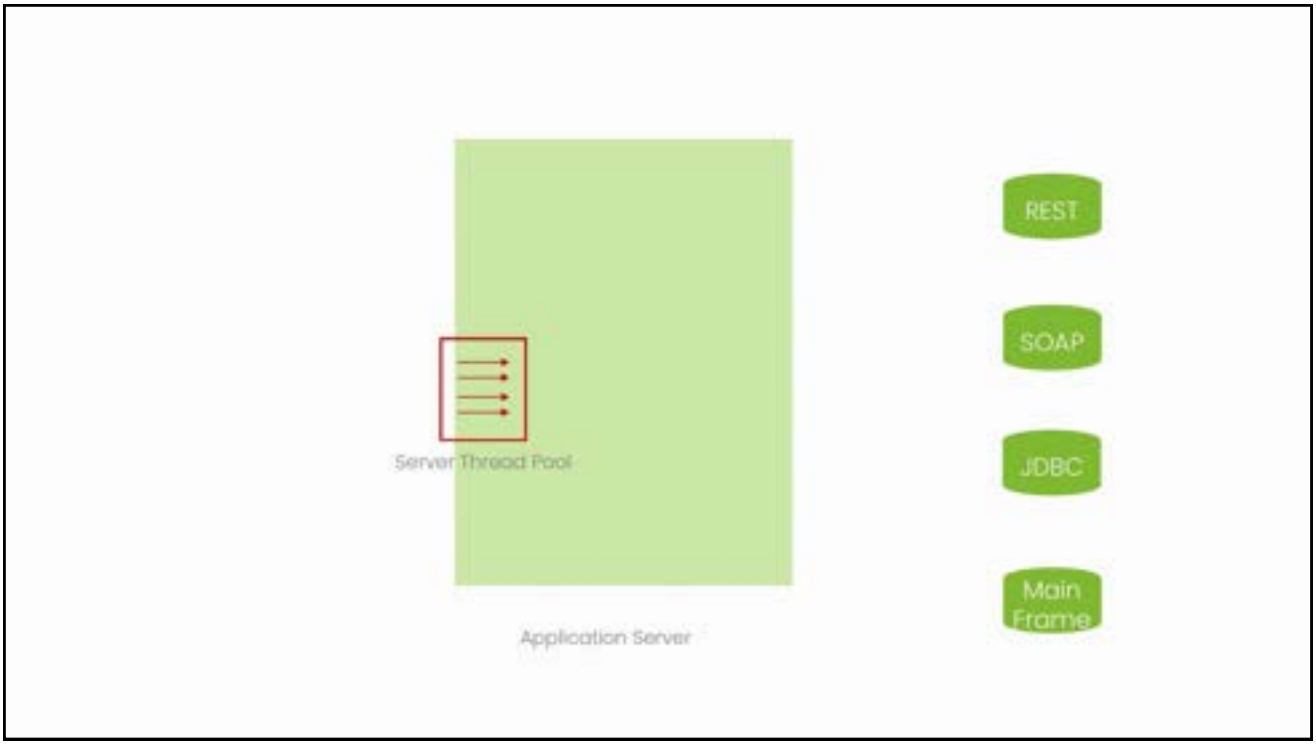


Fig 1: Typical thread's life cycle

5. Once response comes back from the Database, the thread processes it, and sends back the response to customer

6. Thread is returned back to the thread pool

Step #2 to #6 will repeat until the application is shutdown.

If you notice, the thread is actually doing real work only in step #3 and #5. In all other steps(i.e., step #1, step #2, step #4, step #6), it is basically waiting(or doing nothing). In most applications, a significant number of threads predominantly waits during most of its lifetime.

Platform threads architecture

In the previous release of JVM(Java Virtual Machine), there was only one type of thread. It's called as 'classic' or 'platform' thread. Whenever a platform thread is created, an operating system thread is allocated to it. Only when the platform thread exits(i.e., dies) the JVM, this operating system thread is free to do other tasks. Until then, it cannot do any other tasks. Basically, there is a 1:1 mapping between a platform thread and an

operating system thread.

According to this architecture, OS thread will be unnecessarily locked down in step #1, step #2, step #4, step #6 of the thread's life cycle, even though it's not doing anything during these steps. Since OS threads are precious and finite resources, it's time is extensively wasted in this platform threads architecture.

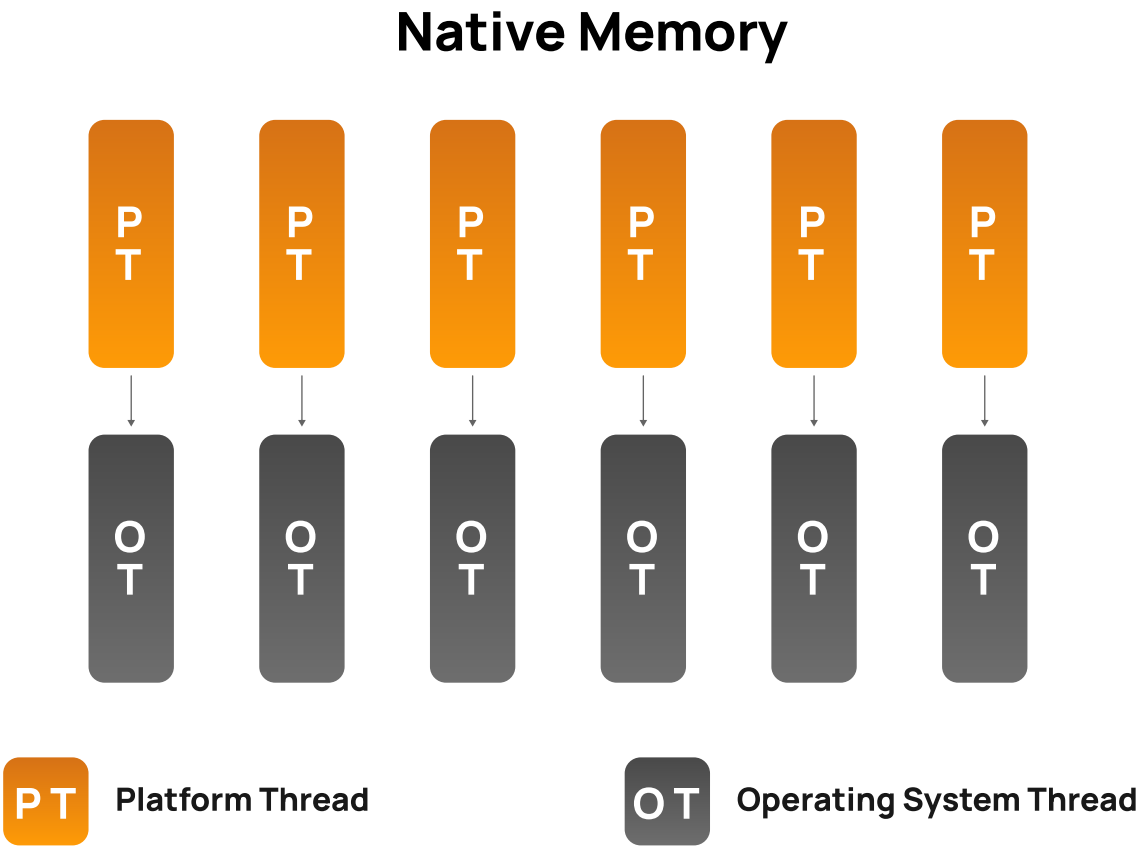


Fig 2: Each Platform Thread is mapped to an Operating System Thread

Virtual threads architecture

In order to efficiently use underlying operating system threads, virtual threads have been introduced in JDK 19. In this new architecture, a virtual thread will be assigned to a platform thread (aka carrier thread) only when it executes real work. As per the above-described thread's life cycle, only during step #3 and step #5 virtual thread will be assigned to the platform thread(which in turn uses OS thread) for execution. In all other steps, virtual thread will be residing as objects in the Java heap memory region just like any of your application objects. Thus, they are lightweight and more efficient.

What is 'stack chunk object'?

When a virtual thread is not executing a real work and resides in a Java heap memory region, it is called as 'stack chunk object'.

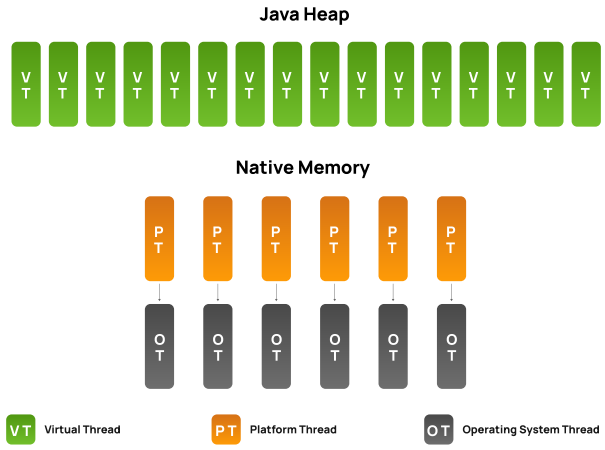


Fig 3: OS Threads are not allocated to Platform threads until real work needs to be executed

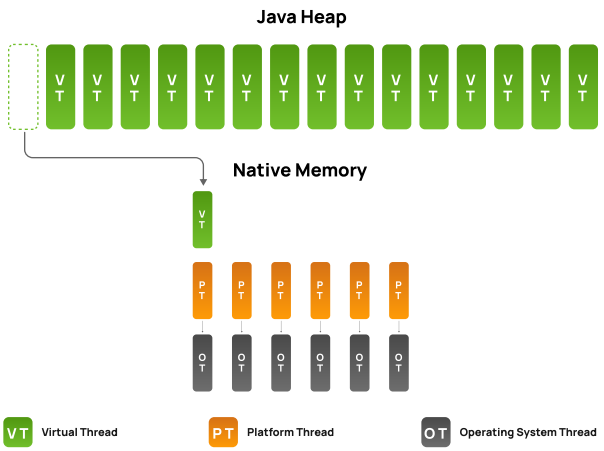


Fig 3.1: Virtual Threads are mapped to platform threads when it does real work

How to create virtual threads?

All the APIs that work with current platform threads, will work with virtual threads as is. However, the APIs to create java virtual threads are slightly different. Below is a sample program that creates Java virtual thread:

```
1:
2: Runnable task = () → { System.out.println("Hello Virtual Thread!"); };
3:
4: Thread.startVirtualThread(task);
```

In this sample program, the virtual thread is created using 'Thread.startVirtualThread()' API. This is the easiest API to create Java virtual threads. If you notice, in the above program in line #4, we are invoking 'Thread.startVirtualThread()' method by passing a Runnable object as an argument, which is created in line #2.

You can also create java virtual threads using the following APIs:

1. Thread.ofVirtual().start(Runnable);

2. Thread.ofVirtual().unstarted(Runnable);

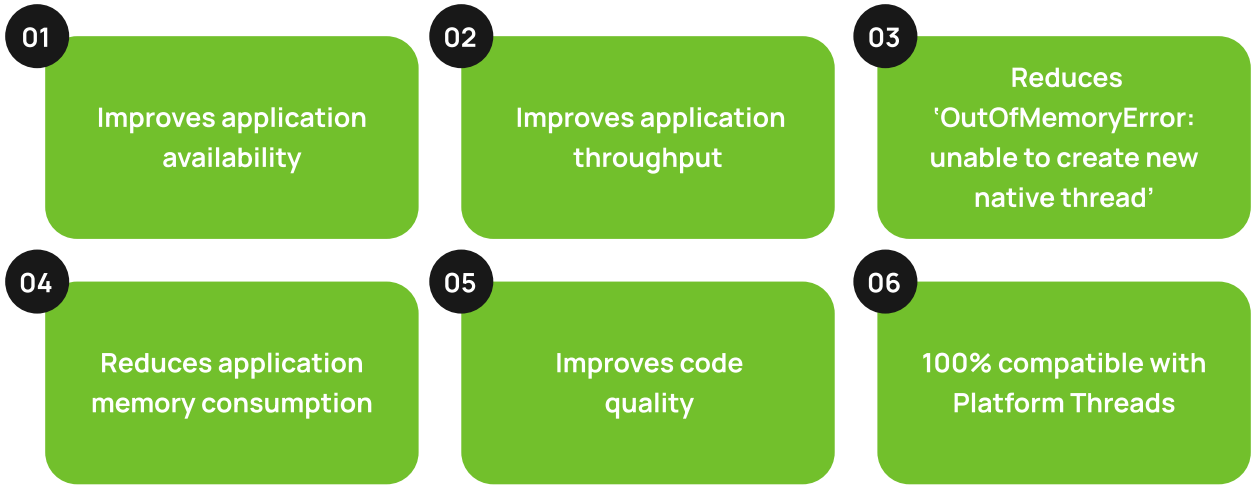
3. Executors.newVirtualThreadPerTaskExecutor();

4. Executors.newThreadPerTaskExecutor(ThreadFactory);

More details about these APIs can be found in this post.

Advantages of Java virtual threads

Because of it's elegant architecture, virtual threads provides several advantages:



To learn more about these advantages in detail, you may read this post.

What is the performance impact of virtual threads?

Java virtual threads are much more lightweight than platform threads. To learn more about java virtual threads performance impact, you may refer to studies done here. However, in a nutshell:

1. If your application either have lot of threads or large stack size (i.e. -Xss), then switching to virtual threads would reduce your application's memory consumption.
2. If your application is creating new threads frequently (instead of leveraging thread pool), switching to virtual threads can improve your application's response time.

Conclusion

We hope this post helped to gain better understanding about java virtual threads – a wonderful addition to our phenomenal Java programming language.

15. Look for exceptions, errors in thread dumps

Thread dumps are vital artifacts to troubleshoot/debug production problems. In the past we have discussed several effective thread dump troubleshooting patterns like: traffic jam, treadmill, RSI, all roads lead to rome ……. In this article we would like to introduce one more thread dump troubleshooting pattern.

How to capture thread dumps?
There are 8 different options to capture thread dumps. You can use the option that is convenient to you.

Thread dumps tend to contain Exceptions or Errors in the threads stack trace. The threads that contain Exceptions or Errors in their stack trace should be investigated. Because they indicate the origin of the problem.

Recently an application was throwing java.lang.OutOfMemoryError. Thread dump was captured from this application. When we analyzed the thread dump, we could notice a particular thread to be throwing java.lang.OutOfMemoryError:

```
Thread 0x3ff781e764e0
 at java.lang.OutOfMemoryError.<init>()V (OutOfMemoryError.java:48)
 at java.lang.ClassLoader.defineClass1(Ljava/lang/String;[BIILjava/security/ProtectionDomain;Ljava/lang/String;)Ljava/lang/
Class; (Native Method)
 at java.lang.ClassLoader.defineClass(Ljava/lang/String;[BIILjava/security/ProtectionDomain;)Ljava/lang/Class;
(ClassLoader.java:757)
 at java.lang.ClassLoader.defineClass(Ljava/lang/String;[BII)Ljava/lang/Class; (ClassLoader.java:636)
 at sun.reflect.GeneratedMethodAccessor37.invoke(Ljava/lang/Object;[Ljava/lang/Object;)Ljava/lang/Object; (Unknown
Source)
 at sun.reflect.DelegatingMethodAccessorImpl.invoke(Ljava/lang/Object;[Ljava/lang/Object;)Ljava/lang/Object;
(DelegatingMethodAccessorImpl.java:43)
 at java.lang.reflect.Method.invoke(Ljava/lang/Object;[Ljava/lang/Object;)Ljava/lang/Object; (Method.java:498)
 at com.sun.xml.bind.v2.runtime.reflect.opt.Injector.inject(Ljava/lang/String;[B)Ljava/lang/Class; (Injector.java:125)
 at com.sun.xml.bind.v2.runtime.reflect.opt.Injector.inject(Ljava/lang/ClassLoader;Ljava/lang/String;[B)Ljava/lang/Class;
(Injector.java:48)
 at com.sun.xml.bind.v2.runtime.reflect.opt.AccessorInjector.prepare(Ljava/lang/Class;Ljava/lang/String;Ljava/lang/String;
[Ljava/lang/String;)Ljava/lang/Class; (AccessorInjector.java:51)
 at com.sun.xml.bind.v2.runtime.reflect.opt.OptimizedAccessorFactory.get(Ljava/lang/reflect/Field;)Lcom/sun/xml/bind/v2/
runtime/reflect/Accessor; (OptimizedAccessorFactory.java:128)
 at com.sun.xml.bind.v2.runtime.reflect.Accessor$FieldReflection.optimize(Lcom/sun/xml/bind/v2/runtime/
JAXBContextImpl;)Lcom/sun/xml/bind/v2/runtime/reflect/Accessor; (Accessor.java:213)
 at com.sun.xml.bind.v2.runtime.reflect.TransducedAccessor$CompositeTransducedAccessorImpl.<init>(Lcom/sun/xml/bind/
v2/runtime/JAXBContextImpl;Lcom/sun/xml/bind/v2/runtime/Transducer;Lcom/sun/xml/bind/v2/runtime/reflect/Accessor;)V
(TransducedAccessor.java:195)
 :
 :
 at com.sun.xml.ws.client.WSServiceDelegate.getPort(Ljavax/xml/namespace/QName;Ljava/lang/Class;[Ljavax/xml/ws/
WebServiceFeature;)Ljava/lang/Object; (WSServiceDelegate.java:274)
 at com.sun.xml.ws.client.WSServiceDelegate.getPort(Ljavax/xml/namespace/QName;Ljava/lang/Class;)Ljava/lang/Object;
(WSServiceDelegate.java:267)
```

From this stacktrace we were able to figure out that this thread is experiencing OutOfMemoryError when it's trying to transform xml into java objects.

Apparently sufficient memory wasn't allocated to this application to process large size XML payloads. Thus when large size XMLs were sent to this application, it started throwing OutOfMemoryError. When sufficient memory was allocated (i.e. increasing -Xmx value), the problem got resolved. Thus looking for Exception or Errors in the thread dumps is a good pattern to identify the root cause of the problem.

But looking for exceptions or errors in a thread dump is not a trivial thing. Because thread dumps tend to contain hundreds or thousands of threads. Each thread will have several lines of stack trace. Going through each line of stack trace to spot exceptions or errors is a tedious process. This where thread dumps analysis tools comes handy. You might consider using free thread dump analysis tools like: fastThread, IBM TDMA, Samurai, … to analyze your application thread dumps.

When you upload thread dump to the fastThread application, it generates a root cause analysis report. One of the sections in this report is 'Exception'. In this section fastThread application reports all the threads that are throwing Exceptions or Errors. Below is the screenshot of this section:

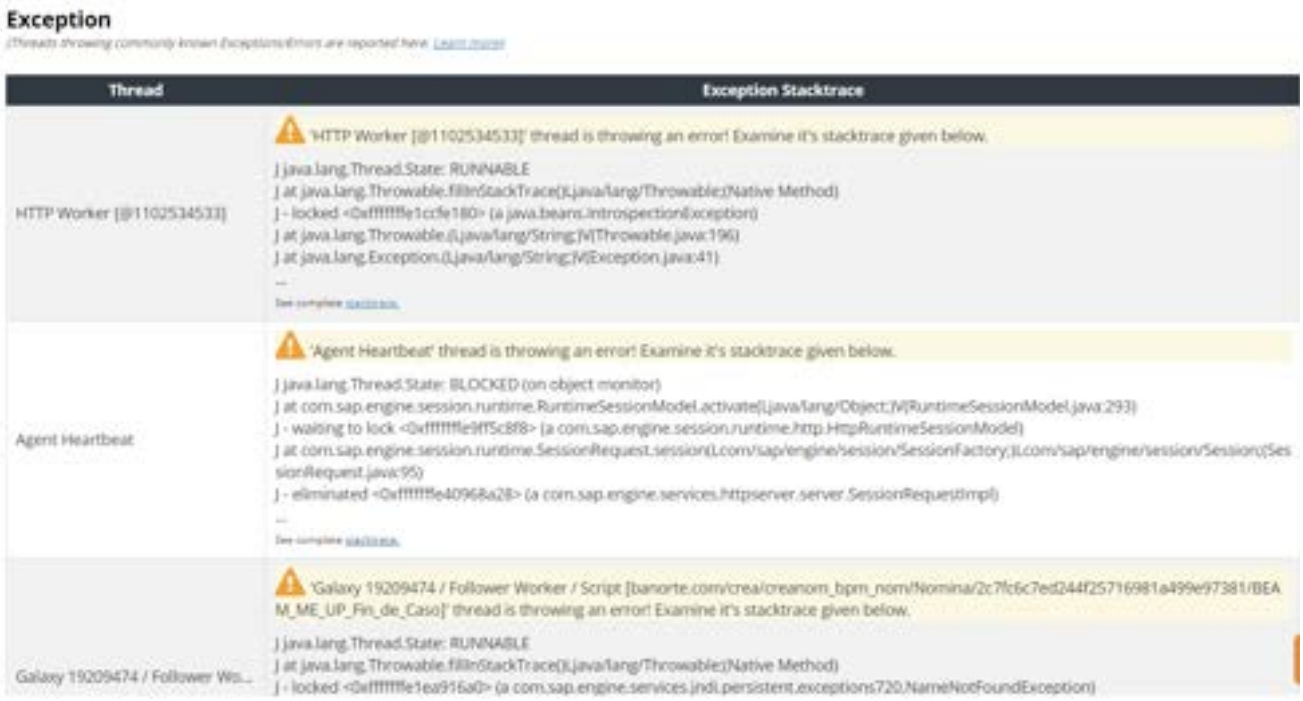


Fig 1: 'Exception' section in fastThread report

You can notice this section is reporting all the threads that have Exceptions or Errors in their stack trace. If any threads are reported in this section, you should consider investigating those thread stack traces to identify the origin of the problem.

16. Thread dump analysis pattern – TREADMILL

Description

You might have experienced the application's CPU to spike up suddenly & spike wouldn't go down until JVM is recycled. You restart the JVM, after certain time period CPU consumption would once again start to spike up. Then you will have to recycle the JVM once again. Have you experienced it? If you have smile on your face now, then it's certain you would have experienced this problem.

This type of problem typically happens when thread spins on an infinite loop. A thread would be spinning infinitely when one of the issues described in this article happens.

To diagnose these sort of problems, you would have to capture 3 thread dumps in an interval of 10 seconds. In between those thread dumps, if there are threads

a. on the same method (or one the same line of code)

b. they are in 'RUNNABLE' state,

then those are the threads which are causing CPU to spike up. Investigating the stack trace of those threads will tell the exact method (or line of code), where threads are spinning. Fixing that particular method (or line of code) would resolve the problem.

Example

HashMap isn't threaded safe implementation. When multiple threads try to access HashMap's get() and put() APIs concurrently it would cause threads go into infinite looping. This problem doesn't happen frequently, but it does happen.

Below is an excerpt from a thread dump which indicates the infinite looping that is happening in HashMap:

```
"Thread-0";quot; prio=6 tid=0x000000000b583000 nid=0x10adc runnable [0x000000000cb6f000]
   java.lang.Thread.State: RUNNABLE
    at java.util.HashMap.put(HashMap.java:374)
    at com.tier1app.HashMapLooper$AddForeverThread.AddForever(NonTerminatingLooper.java:32)
    at com.tier1app.HashMapLooper$AddForeverThread.method2(NonTerminatingLooper.java:27)
    at com.tier1app.HashMapLooper$AddForeverThread.method1(NonTerminatingLooper.java:22)
    at com.tier1app.NonTerminatingLooper$LoopForeverThread.run(NonTerminatingLooper.java:16)
```

Across all the 3 thread dumps “Thread-0” was always exhibiting same stack trace. i.e. it was always in the java.util.HashMap.put(HashMap.java:374) method. This problem was addressed by replacing the HashMap with ConcurrentHashMap.

Why named as Treadmill?

In Treadmill, one would keep running without moving forward. Similarly, when there is infinite looping, CPU consumption goes high without progress in the code execution path. Thus it’s named as ‘Treadmill’ pattern.

17. Thread dump analysis pattern – ATHLETE

Description

Threads in 'runnable' state consume CPU. So when you are analyzing thread dumps for **high CPU consumption**, threads in 'runnable' state should be thoroughly reviewed.Typically in thread dumps several threads are classified in 'RUNNABLE' state. But in reality several of them wouldn't be actually running, rather they would be just waiting. But still, JVM classifies them in 'RUNNABLE' state. You need to learn to differentiate from really running threads with pretending/misleading **RUNNABLE threads**.

Example

Below is the real world thread dump excerpt. In these stack traces, threads aren't actually in 'RUNNABLE' state. i.e. they are not actively executing any code. They are just waiting on the sockets to read or write. It's because JVM doesn't really know what thread is doing in a native method, JVM classifies them in 'RUNNABLE' state. Real running threads would consume CPU, whereas these threads are on I/O wait, which don't consume any CPU.

```
WorkerThread#27[10.201.1.55:56893] - priority:10 - threadId:0x00002b59983cb000 - nativeId:0x4482 - addressSpace:null - state:RUNNABLE
stackTrace:
 java.lang.Thread.State: RUNNABLE
 at java.net.SocketInputStream.socketRead0(Native Method)
 at java.net.SocketInputStream.___AW_read(SocketInputStream.java:129)
 at java.net.SocketInputStream.read(SocketInputStream.java)
 at java.io.BufferedInputStream.fill(BufferedInputStream.java:218)
 at java.io.BufferedInputStream.read(BufferedInputStream.java:237)
 - locked  (a java.io.BufferedInputStream)
 at java.io.DataInputStream.readByte(DataInputStream.java:248)
 at org.jboss.jms.server.remoting.ServerSocketWrapper.checkConnection(ServerSocketWrapper.java:94)
 at org.jboss.remoting.transport.socket.ServerThread.acknowledge(ServerThread.java:857)
 at org.jboss.remoting.transport.socket.ServerThread.dorun(ServerThread.java:585)
 at org.jboss.remoting.transport.socket.ServerThread.run(ServerThread.java:234)
```

```
multicast receiver,GATES-RefData-Infinispan-Cluster,matcamupa67-vm-46719 - priority:10 - threadId:0x00002b597c1ba800 - nativeId:0x3034 - addressSpace:null - state:RUNNABLE
stackTrace:
 java.lang.Thread.State: RUNNABLE
 at java.net.PlainDatagramSocketImpl.receive0(Native Method)
 - locked  (a java.net.PlainDatagramSocketImpl)
 at java.net.PlainDatagramSocketImpl.receive(PlainDatagramSocketImpl.java:145)
 - locked  (a java.net.PlainDatagramSocketImpl)
 at java.net.DatagramSocket.receive(DatagramSocket.java:725)
 - locked  (a java.net.DatagramPacket)
 - locked  (a java.net.MulticastSocket)
 at org.jgroups.protocols.UDP$PacketReceiver.___AW_run(UDP.java:675)
 at org.jgroups.protocols.UDP$PacketReceiver.run(UDP.java)
 at java.lang.Thread.run(Thread.java:662
```

Why named as Athlete Pattern?

Wikipedia defines Athlete as a person who competes in one or more sports that involve physical strength, speed, and/or endurance. Athlete consumes high energy, similarly really RUNNABLE threads consumes high CPU, whereas pretending RUNNABLE threads don't.

18. Thread dump analysis pattern –ATHEROSCLEROSIS

Description

If threads are blocking momentarily, then it's not a problem. However, if they are blocking for a prolonged period, then it's of concern. It's indicative of some problem in the application. Typically blocked threads would make application unresponsive.

Threads that remain in the same method and in 'BLOCKED' state between 3 threads dump which are captured in an interval of 10 seconds can turn out to be problematic ones. Studying the stack traces of such blocked threads would indicate the reasons why they are blocked. Reasons may include: deadlocks, circular deadlocks, another thread, would have acquired the locked and never released it, external SORs could have become unresponsive …

Example

Following is the excerpt of a thread dump that was captured from a major SOA application, which became unresponsive. The thread **ajp-bio-192.168.100.128-9022-exec-173** remained in BLOCKED state for 3 consecutive thread dumps which were captured in an interval of 10 seconds. Here goes the important parts of Stack trace of this thread:

```
ajp-bio-192.168.100.128-9022-exec-173
Stack Trace is:
java.lang.Thread.State: BLOCKED (on object monitor)
at
**.***.sp.dao.impl.ReferenceNumberGeneratorDaoImpl.getNextItineryReferenceNumber(ReferenceNumberGeneratorDaoImpl.j
ava:55)
- waiting to lock 0x00000006afaa5a60 (a **.***.sp.dao.impl.ReferenceNumberGeneratorDaoImpl)
```

```
at sun.reflect.GeneratedMethodAccessor3112.invoke(Unknown Source)
at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
at java.lang.reflect.Method.invoke(Method.java:601)
at org.springframework.aop.support.AopUtils.invokeJoinpointUsingReflection(AopUtils.java:307)
:
:
at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1145)
at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:615)
at java.lang.Thread.run(Thread.java:722)
```

Here you can notice that ajp-bio-192.168.100.128-9022-exec-173 stuck in the method **.***.sp.dao.impl.ReferenceNumberGeneratorDaoImpl.getNextItineryReferenceNumber(ReferenceNumberGeneratorDaoImpl.java:55). This thread got stuck in this method, because another thread ajp-bio-192.168.100.128-9022-exec-84 after obtaining the lock 0x00000006afaa5a60, it never returned back. Below goes the stack trace of ajp-bio-192.168.100.128-9022-exec-84 thread

```
ajp-bio-192.168.100.128-9022-exec-84
Stack Trace is:
java.lang.Thread.State: RUNNABLE
at java.net.SocketInputStream.socketRead0(Native Method)
at java.net.SocketInputStream.read(SocketInputStream.java:150)
at java.net.SocketInputStream.read(SocketInputStream.java:121)
at java.io.DataInputStream.readFully(DataInputStream.java:195)
at java.io.DataInputStream.readFully(DataInputStream.java:169)
at net.sourceforge.jtds.jdbc.SharedSocket.readPacket(SharedSocket.java:850)
at net.sourceforge.jtds.jdbc.SharedSocket.getNetPacket(SharedSocket.java:731)
- locked 0x00000006afa88a68 (a java.util.concurrent.ConcurrentHashMap)
at net.sourceforge.jtds.jdbc.ResponseStream.getPacket(ResponseStream.java:477)
at net.sourceforge.jtds.jdbc.ResponseStream.read(ResponseStream.java:114)
at net.sourceforge.jtds.jdbc.ResponseStream.peek(ResponseStream.java:99)
at net.sourceforge.jtds.jdbc.TdsCore.wait(TdsCore.java:4127)
at net.sourceforge.jtds.jdbc.TdsCore.executeSQL(TdsCore.java:1086)
- locked 0x00000006bca709f8 (a net.sourceforge.jtds.jdbc.TdsCore)
at net.sourceforge.jtds.jdbc.JtdsStatement.executeSQLQuery(JtdsStatement.java:493)
at net.sourceforge.jtds.jdbc.JtdsPreparedStatement.executeQuery(JtdsPreparedStatement.java:1032)
at com.jolbox.bonecp.PreparedStatementHandle.executeQuery(PreparedStatementHandle.java:174)
at
**.***.sp.dao.impl.ReferenceNumberGeneratorDaoImpl.getNextItineryReferenceNumber(ReferenceNumberGeneratorDaoImpl.j
ava:65)
- locked 0x00000006afaa5a60 (a **.***.sp.dao.impl.ReferenceNumberGeneratorDaoImpl)
at sun.reflect.GeneratedMethodAccessor3112.invoke(Unknown Source)
at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
at java.lang.reflect.Method.invoke(Method.java:601)
at org.springframework.aop.support.AopUtils.invokeJoinpointUsingReflection(AopUtils.java:307)
:
:
```

```
at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1145)
at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:615)
at java.lang.Thread.run(Thread.java:722)
```

Apparently, it turned out that JDBC read timeout was never set, due to that ajp-bio-192.168.100.128-9022-exec-84 never returned back from it's JDBC call. Thus it blocked all other threads which were invoking **.***.sp.dao.impl.ReferenceNumberGeneratorDaoImpl.getNextItineryReferenceNumber() method.

Why named as Atherosclerosis?

Atherosclerosis is a heart disease. Medically it's defined as the following: the inside walls of human arteries are normally smooth and flexible, allowing blood to flow through them easily. Fatty deposits, or plaques, may build up inside the arterial wall. These plaques narrow the artery and can reduce or even completely stop the flow of blood, leading to death.

Similarly if blocking of a thread prolongs and happens across multiple threads, then it would make the application unresponsive, eventually, it has to be killed.

19. Thread dump analysis pattern – REPETITIVE STRAIN INJURY (RSI)

Description

When there is a performance bottleneck in the application, most of the threads will start to accumulate on that problematic bottleneck area. Those threads will have same stack trace. Thus whenever a significant number of threads exhibit identical/repetitive stack trace then those stack trace should be investigated. It may be indicative of performance problems.

Here are few such scenarios:

Scenario 1

Say your SOR or external service is slowing down then a significant number of threads will start to wait for its response. In such circumstance, those threads will exhibit same stack trace.

Scenario 2

Say a thread acquired a lock & it never released then, then several other threads which are in the same execution path will get into the blocked state, exhibiting same stack trace.

Scenario 3

If a loop (for loop, while loop, do..while loop) condition doesn't terminate then several threads which execute that loop will exhibit the same stack trace.

When any of the above scenarios occurs application's performance, and availability will be questioned.

Example

Below is an excerpt from a thread dump of a major B2B application. This application was running fine, but all of a sudden it became unresponsive. Thread dump from this application was captured. It revealed that 225 threads out of 400 threads were exhibiting same stack trace. Here goes that stack trace:

```
"ajp-bio-192.168.100.128-9022-exec-79" daemon prio=10 tid=0x00007f4d2001c000 nid=0x1d1c waiting for monitor entry
[0x00007f4ce91fa000]
  java.lang.Thread.State: BLOCKED (on object monitor)
 at
com.xxxxxxxxx.xx.xxx.xxxx.ReferenceNumberGeneratorDaoImpl.getNextItineryReferenceNumber(ReferenceNumberGenerato
rDaoImpl.java:55)
 - waiting to lock 0x00000006afaa5a60 (a com.xxxxxxxxx.sp.dao.impl.ReferenceNumberGeneratorDaoImpl)
 at sun.reflect.GeneratedMethodAccessor3112.invoke(Unknown Source)
 at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
 at java.lang.reflect.Method.invoke(Method.java:601)
 at org.springframework.aop.support.AopUtils.invokeJoinpointUsingReflection(AopUtils.java:307)
 at org.springframework.aop.framework.ReflectiveMethodInvocation.invokeJoinpoint(ReflectiveMethodInvocation.java:182)
 at org.springframework.aop.framework.ReflectiveMethodInvocation.proceed(ReflectiveMethodInvocation.java:149)
 at org.springframework.orm.hibernate3.HibernateInterceptor.invoke(HibernateInterceptor.java:111)
 at org.springframework.aop.framework.ReflectiveMethodInvocation.proceed(ReflectiveMethodInvocation.java:171)
 at org.springframework.aop.framework.JdkDynamicAopProxy.invoke(JdkDynamicAopProxy.java:204)
 at com.sun.proxy.$Proxy36.getNextItineryReferenceNumber(Unknown Source)
 at
com.xxxxxxxxx.xx.xxxxxxxx.xxx.ReferenceNumberGeneratorServiceImpl.getNextItineryReferenceNumber(ReferenceNumberG
eneratorServiceImpl.java:15)
 :
 :
```

From the stack trace, you can infer that thread was blocked and waiting for the lock on the object **0x00000006afaa5a60**. 225 such threads were waiting to obtain lock on this same object. It's definitely a bad sign. It's a clear indication of thread starvation.

Apparently this lock was held by "ajp-bio-192.168.100.128-9022-exec-84". Below is the stack trace this thread. You can notice that this thread acquired the lock on the object **0x00000006afaa5a60**, but after acquiring the lock, it got stuck waiting for response from the database. Apparently for this application database timeout wasn't set. Due to that this thread's database call never returned back. Due to that 225 other threads were stuck. Thus application became unresponsive.

After setting proper database time out value, this problem went away.

```
"ajp-bio-192.168.100.128-9022-exec-84" daemon prio=10 tid=0x00007f4d2000a800 nid=0x1d26 runnable
[0x00007f4ce6ce1000]
   java.lang.Thread.State: RUNNABLE
 at java.net.SocketInputStream.socketRead0(Native Method)
 at java.net.SocketInputStream.read(SocketInputStream.java:150)
 at java.net.SocketInputStream.read(SocketInputStream.java:121)
 at java.io.DataInputStream.readFully(DataInputStream.java:195)
 at java.io.DataInputStream.readFully(DataInputStream.java:169)
 at net.sourceforge.jtds.jdbc.SharedSocket.readPacket(SharedSocket.java:850)
 at net.sourceforge.jtds.jdbc.SharedSocket.getNetPacket(SharedSocket.java:731)
 - locked  0x0000000b8dcc8c81 (a java.util.concurrent.ConcurrentHashMap)
 at net.sourceforge.jtds.jdbc.RexxonseStream.getPacket(RexxonseStream.java:477)
 at net.sourceforge.jtds.jdbc.RexxonseStream.read(RexxonseStream.java:114)
 at net.sourceforge.jtds.jdbc.RexxonseStream.peek(RexxonseStream.java:99)
 at net.sourceforge.jtds.jdbc.TdsCore.wait(TdsCore.java:4127)
 at net.sourceforge.jtds.jdbc.TdsCore.executeSQL(TdsCore.java:1086)
 - locked 0x0000000d1cdd7b17 (a net.sourceforge.jtds.jdbc.TdsCore)
 at net.sourceforge.jtds.jdbc.JtdsStatement.executeSQLQuery(JtdsStatement.java:493)
 at net.sourceforge.jtds.jdbc.JtdxxreparedStatement.executeQuery(JtdxxreparedStatement.java:1032)
 at com.jolbox.bonecp.PreparedStatementHandle.executeQuery(PreparedStatementHandle.java:174)
 at
com.xxxxxxxx.xx.xxx.xxx.ReferenceNumberGeneratorxxxxxx.getNextItineryReferenceNumber(ReferenceNumberGeneratorxx
xxx.java:65)
 - locked  0x00000006afaa5a60(a com.xxxxxxxx.xx.xxx.xxx.ReferenceNumberGeneratorxxxxxx)
 at sun.reflect.GeneratedMethodAccessor3112.invoke(Unknown Source)
 at sun.reflect.DelegatingMethodAccessorxxx.invoke(DelegatingMethodAccessorxxx.java:43)
 at java.lang.reflect.Method.invoke(Method.java:601)
 at org.springframework.aop.support.AopUtils.invokeJoinpointUsingReflection(AopUtils.java:307)
 at org.springframework.aop.framework.ReflectiveMethodInvocation.invokeJoinpoint(ReflectiveMethodInvocation.java:182)
 at org.springframework.aop.framework.ReflectiveMethodInvocation.proceed(ReflectiveMethodInvocation.java:149)
 at org.springframework.orm.hibernate3.HibernateInterceptor.invoke(HibernateInterceptor.java:111)
 at org.springframework.aop.framework.ReflectiveMethodInvocation.proceed(ReflectiveMethodInvocation.java:171)
 at org.springframework.aop.framework.JdkDynamicAopProxy.invoke(JdkDynamicAopProxy.java:204)
 at com.sun.proxy.$Proxy36.getNextItineryReferenceNumber(Unknown Source)
 at
com.xxxxxxxx.xx.service.xxx.ReferenceNumberGeneratorServicexxx.getNextItineryReferenceNumber(ReferenceNumberGene
ratorServicexxx.java:15)
 at sun.reflect.GeneratedMethodAccessor3031.invoke(Unknown Source)
 at sun.reflect.DelegatingMethodAccessorxxx.invoke(DelegatingMethodAccessorxxx.java:43)
 at java.lang.reflect.Method.invoke(Method.java:601)
 at org.springframework.aop.support.AopUtils.invokeJoinpointUsingReflection(AopUtils.java:307)
      :
      :
```

So bottom line is always looking for repeating stack traces

Why named as RSI?

A Repetitive Strain Injury (RSI) happens when you exercise your body parts (hand, fingers, wrist, neck,…) repeatedly in a wrong posture. Similarly when there is a performance bottleneck, multiple threads will start to exhibit the stack trace again & again. Those stack trace should be studied in detail.

20. Thread dump analysis pattern – TRAFFIC JAM

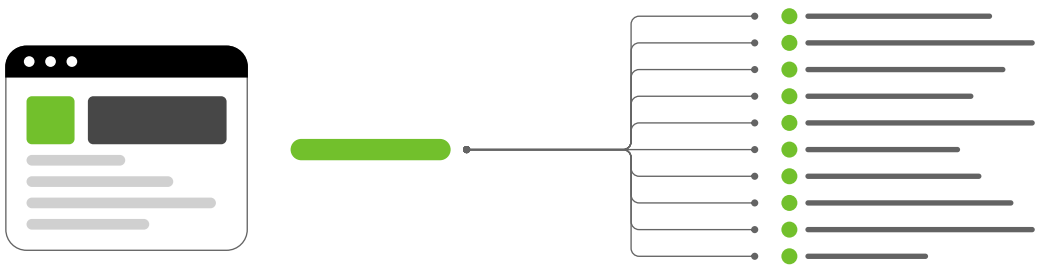
Description

Thread-A could have acquired the lock-1 and then would never release it. Thread-B could have acquired lock-2 and waiting on this lock-1. Thread-C could be waiting to acquire lock-2. This kind of transitive blocks between threads can make entire application unresponsive. See the real-world example below.

Example

Below is a real-world example taken from a major travel application. Here 'Finalizer' thread was waiting for a lock that was held by '**ajp-bio-192.168.100.41-7078-exec-40**' thread. ajp-bio-192.168.100.41-7078-exec-40 and several other threads were waiting for the lock which took place by '**ajp-bio-192.168.100.41-7078-exec-12**' thread.

Thus 'ajp-bio-192.168.100.41-7078-exec-12' has transitively blocked 42 threads in total. This ripple effect caused the entire application to become unresponsive. Apparently, it turned out 'ajp-bio-192.168.100.41-7078-exec-12' was blocked indefinitely because of a bug in an APM monitoring agent (Ruxit). Upgrading the agent version to latest version resolved the problem. This is quite an irony because – APM monitoring agents are meant to prevent/isolate these sort of issues, but in this case, they themselves are causing the issue. It's like a law enforcement breaking the laws.



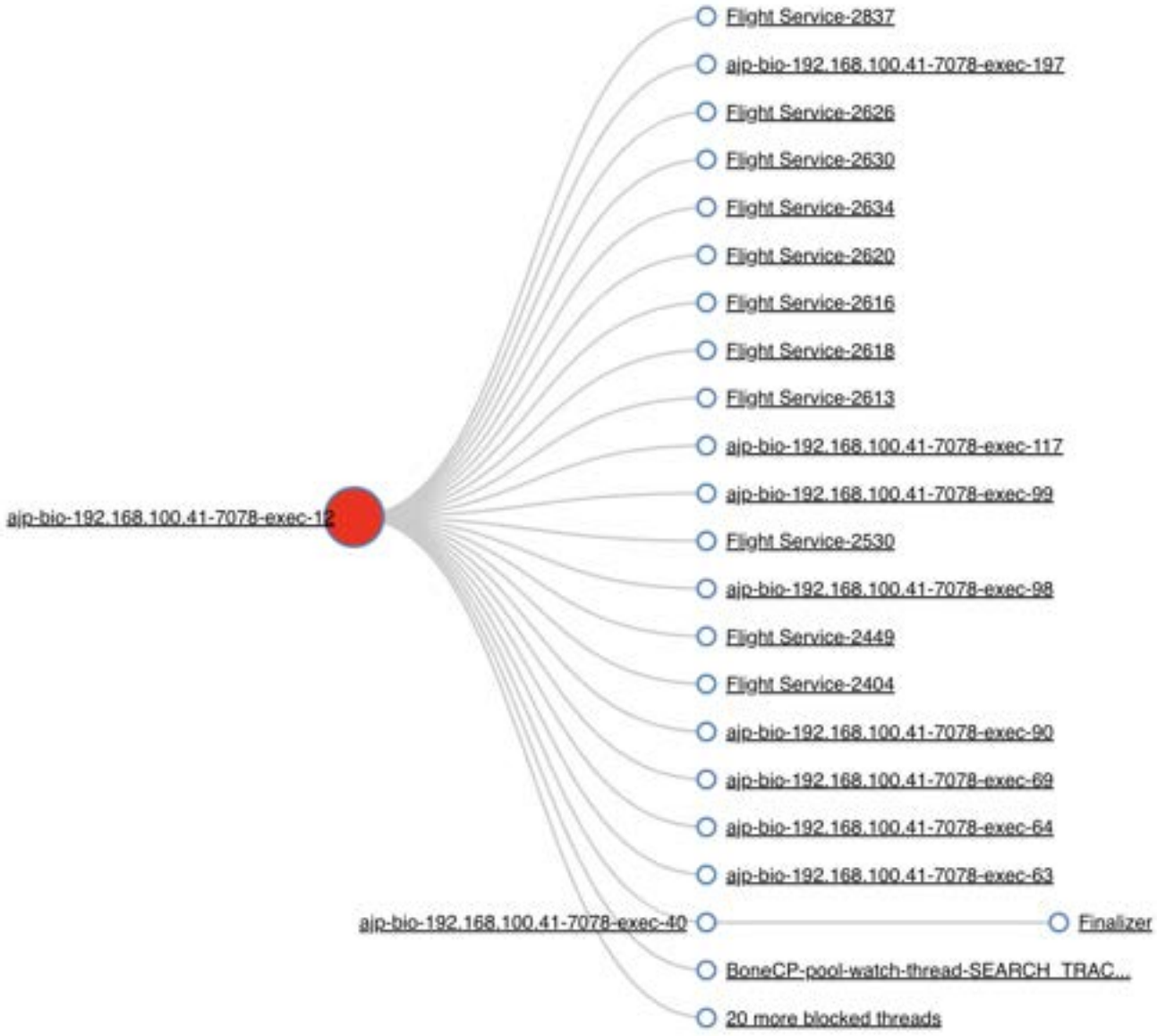


Fig 1: Graph showing transitive blocks among threads

Please refer to this document for the stack trace of the blocked threads.

Why named as Traffic Jam?

Traffic Jam typically happens when there is an accident in the front. Due to that, all the cars that are following the front car will also get stranded. This is very analogous to the transitive blocks behavior described here.

21. Thread dump analysis pattern – ALL ROADS LEAD TO ROME

Description

If several threads in a thread dump end up working on one single method, then it might be of concern. Most of the times, if there is a problem (say poorly responding data source, un-relinquished lock, infinite looping threads …), then a significant number of threads will end up in one single method. That particular method has to be analyzed in detail.

Example

This application was connecting with Apache Cassandra NoSQL Database. The application uses DataStax java driver to connect with Cassandra. DataStax has a dependency on the netty library. To be specific following are the libraries that application uses:

cassandra-driver-core-2.0.1.jar

netty-3.9.0.Final.jar

This application all of sudden ran into 'java.lang.OutOfMemoryError: unable to create new native thread'. When thread dump was taken on the application, around 2460 threads were in 'runnable' state stuck in the method: **sun.nio.ch.EPollArrayWrapper.epollWait(Native Method)**. Below is the stack trace of one of the thread:

```
"New I/O worker #211" prio=10 tid=0x00007fa06424d000 nid=0x1a58 runnable [0x00007f9f832f6000]
  java.lang.Thread.State: RUNNABLE
at sun.nio.ch.EPollArrayWrapper.epollWait(Native Method)
at sun.nio.ch.EPollArrayWrapper.poll(EPollArrayWrapper.java:228)
at sun.nio.ch.EPollSelectorImpl.doSelect(EPollSelectorImpl.java:81)
at sun.nio.ch.SelectorImpl.lockAndDoSelect(SelectorImpl.java:87)
- locked  (a sun.nio.ch.Util$2)
- locked  (a java.util.Collections$UnmodifiableSet)
- locked  (a sun.nio.ch.EPollSelectorImpl)
at sun.nio.ch.SelectorImpl.select(SelectorImpl.java:98)
at org.jboss.netty.channel.socket.nio.SelectorUtil.select(SelectorUtil.java:68)
at org.jboss.netty.channel.socket.nio.AbstractNioSelector.select(AbstractNioSelector.java:415)
at org.jboss.netty.channel.socket.nio.AbstractNioSelector.run(AbstractNioSelector.java:212)
at org.jboss.netty.channel.socket.nio.AbstractNioWorker.run(AbstractNioWorker.java:89)
at org.jboss.netty.channel.socket.nio.NioWorker.run(NioWorker.java:178)
at org.jboss.netty.util.ThreadRenamingRunnable.run(ThreadRenamingRunnable.java:108)
at org.jboss.netty.util.internal.DeadLockProofWorker$1.run(DeadLockProofWorker.java:42)
at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1145)
at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:615)
at java.lang.Thread.run(Thread.java:722)
```

Oh boy! This is way too many threads. All of them are netty library threads. Apparently, the issue turned out that Cassandra NoSQL DB ran out of space. This issue was cascading in the application as OutOfMemoryError. When more space was allocated in Cassandra DB, the problem went away.

Thus always look out for the method(s) where most of the threads are working.

Why named as 'All roads lead to Rome'?

'All Roads lead to Rome' is a famous proverb to indicate different paths can take to one final end. Similarly, when there is a problem, there is a high chance that several threads will finally end up in one problematic method.

22. Thread dump analysis pattern – ADDITIVES

Description

It's highly recommended to capture 3 threads dumps in an interval of 10 seconds to uncover any problem in the JVM. If in the 2nd and 3rd thread dump if additional threads start to go into a particular state, then those threads and their stack traces have to be studied in detail. It may or may not be indicative of certain problem in the application, but definitely, a good lead to follow.

Example

This problem surfaced because of a thread leak in Oracle JDBC Driver when ONS feature was turned ON. This problem happened in an old version of Oracle JDBC Driver (almost in 2011). Because of the bug in the driver, under certain scenarios, it started to spawn tonnes of new threads. In every captured thread dump new threads in RUNNABLE stated with below stack trace got added. Around 1700 threads with the same stack trace got created.

```
Thread-6805 - priority:8 - threadId:0x07768000 - nativeId:10966 - addressSpace:null - state:RUNNABLE
stackTrace:
 java.lang.Thread.State: RUNNABLE
 at java.net.SocketInputStream.socketRead0(Native Method)
 at java.net.SocketInputStream.read(SocketInputStream.java:155)
 at java.net.SocketInputStream.read(SocketInputStream.java:121)
 at oracle.ons.InputBuffer.readMoreData(InputBuffer.java:268)
 at oracle.ons.InputBuffer.getNextString(InputBuffer.java:223)
 at oracle.ons.ReceiverThread.run(ReceiverThread.java:266)
```

Why named as additives?

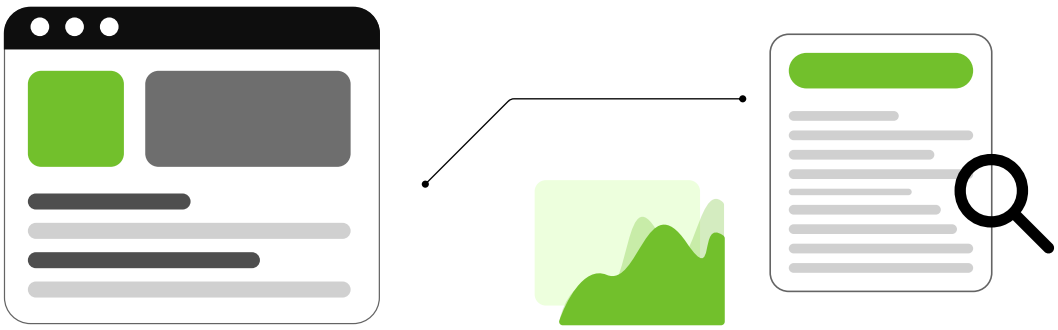
Additives are defined as 'a substance added to something in small quantities'. Similarly, this pattern talks above addition of new threads to an existing state.

23. Thread dump analysis pattern – LEPRECHAUN TRAP

Description

Objects that have finalize() method are treated differently during Garbage collection process than the ones which don't have them. During garbage collection phase, object with finalize() aren't immediately evicted from the memory. Instead as first step, those objects are added to an internal queue of java.lang.ref.Finalizer object. There is a low priority JVM thread by name 'Finalizer' that executes finalize() method of each object in the queue. Only after the execution of finalize() method, object becomes eligible for Garbage Collection. Because of poor implementation of finalize() method if Finalizer thread gets blocked then it will have a severe detrimental cascading effect on the JVM.

If Finalizer thread gets blocked then internal queue of java.lang.ref.Finalize will start to grow. It would cause JVM's memory consumption to grow rapidly. It would result in OutOfMemoryError, jeopardizing entire JVM's availability. Thus when analyzing thread dumps it's highly recommended to study the stack trace of Finalizer thread.



Example

Here is a sample stack trace of a Finalizer thread which got blocked in a finalize() meth

```
"Finalizer" daemon prio=10 tid=0x00007fb2dc32b000 nid=0x7a21 waiting for monitor entry [0x00007fb2cdcb6000]
  java.lang.Thread.State: BLOCKED (on object monitor)
at net.sourceforge.jtds.jdbc.JtdsConnection.releaseTds(JtdsConnection.java:2024)
- waiting to lock 0x00000007d50d98f0 (a net.sourceforge.jtds.jdbc.JtdsConnection)
at net.sourceforge.jtds.jdbc.JtdsStatement.close(JtdsStatement.java:972)
at net.sourceforge.jtds.jdbc.JtdsStatement.finalize(JtdsStatement.java:219)
at java.lang.ref.Finalizer.invokeFinalizeMethod(Native Method)
at java.lang.ref.Finalizer.runFinalizer(Finalizer.java:101)
at java.lang.ref.Finalizer.access$100(Finalizer.java:32)
at java.lang.ref.Finalizer$FinalizerThread.run(Finalizer.java:178)
```

Above stack trace was captured from a JVM which was using one of the older versions of JTDS JDBC Driver. Apparently this version of driver had an issue; you can see finalize() method in the net.sourceforge.jtds.jdbc.JtdsStatement object calling JtdsConnection#releaseTds() method. Apparently, this method got blocked and never returned back. Thus Finalizer thread got stuck indefinitely in the JtdsConnection#releaseTds() method. Due to that Finalizer thread wasn't able to work on the other objects that had finalize() method. Due to that application started to suffer from OutOfMemoryError. In the latest version of JTDS JDBC Driver this issue was fixed. Thus when you are implementing finalize() method be very careful.

Video Tutorial

Here is a good video tutorial that explains this pattern visually:

Watch video

https://www.youtube.com/watch?v=UjP2081groY

Why named as Leprechaun Trap?

Kids in western countries build Leprechaun Trap as part of St. Patrick's day celebration. Leprechaun is a fairy character, basically a very tiny old man, wearing a green coat & hat who is in search for gold coins. Kids build creative traps for this Laprechaun, luring him with gold coins. Similarly anxious Finalizer thread is always in search of objects that has finalize() method to execute them. In case if finalize() method is wrongly implemented, it can trap the Finalizer thread. Because of this similarity we have named it as Leprechaun Trap.

24. Thread dump analysis pattern – SEVERAL SCAVENGERS

Description

Based on the type of GC algorithm (Serial, parallel, G1, CMS) used, default number of garbage collection threads gets created. Details on default number of threads that will be created are documented below. Sometimes too many extraneous GC threads would get created based on the default configuration. We have seen scenarios where 128, 256, 512 GC threads got created based on default configuration. Too many GC threads can also affect your application's performance. So GC thread count should be carefully configured.

Parallel GC

If you are using Parallel GC algorithm, then number of GC threads is controlled by -XX:ParallelGCThreads property. Default value for -XX:ParallelGCThreads on Linux/x86 machine is derived based on the formula:

```
if (num of processors < =8) {
  return num of processors; } else {
 return 8+(num of processors-8)*(5/8);
}
```

So if your JVM is running on server with 32 processors, then ParallelGCThread value is going to be: 23(i.e. 8 + (32 – 8)*(5/8)).

CMS GC

If you are using CMS GC algorithm, then number of GC threads is controlled by -**XX:ParallelGCThreads** and -**XX:ConcGCThreads** properties. Default value of -**XX:ConcGCThreads** is derived based on the formula:

max((ParallelGCThreads+2)/4, 1)

So if your JVM is running on server with 32 processors, then

- ParallelGCThread value is going to be: 23 (i.e. 8 + (32 – 8)*(5/8))
- ConcGCThreads value is going to be: 6.
- So total GC thread count is: **29** (i.e. ParallelGCThread count + ConcGCThreads i.e. 23 + 6)

G1 GC

IIf you are using G1 GC algorithm, then number of GC threads is controlled by -**XX:ParallelGCThreads**, -**XX:ConcGCThreads**, -**XX:G1ConcRefinementThreads** properties. Default value of -**XX:G1ConcRefinementThreads** is derived based on the formula:

ParallelGCThreads+1

So if your JVM is running on server with 32 processors, then

- ParallelGCThread value is going to be: 23 (i.e. 8 + (32 – 8)*(5/8))
- ConcGCThreads value is going to be: 6
- G1ConcRefinementThreads value is going to be 24 (i.e. 23 + 1)
- So total GC thread count is: **53** (i.e. ParallelGCThread count + ConcGCThreads + G1ConcRefinementThreads i.e. 23 + 6 + 24)

53 threads for GC is quite a high number. It should be tuned down appropriately.

Why named as Several Scavengers?

Scavenger is a person who searches for and cleans-up discarded items. GC threads also do exactly same thing. Too many Scavengers in one single spot doesn't yield productive result. Similarly, too many GC threads wouldn't help JVM either.