Fehlerbehebung bei Java Stack High CPU Utilization

Inhalt

Einführung

Dieses Dokument beschreibt Java Stack (Jstack) und dessen Verwendung, um die Ursache der hohen CPU-Auslastung in Cisco Policy Suite (CPS) zu ermitteln.

Fehlerbehebung mit Jstack

Was ist Jstack?

Jstack nimmt einen Speicherabdruck eines laufenden Java-Prozesses (in CPS ist QNS ein Java-Prozess). Jstack enthält alle Details dieses Java-Prozesses, z. B. Threads/Anwendungen und die Funktionalität jedes Threads.

Warum benötigen Sie Jstack?

Jstack stellt die Jstack-Ablaufverfolgung bereit, sodass Ingenieure und Entwickler den Zustand jedes Threads kennen können.

Der Linux-Befehl zum Abrufen der Jstack-Nachverfolgung des Java-Prozesses lautet:

```
# jstack <process id of Java process>
```

Der Speicherort des Jstack-Prozesses in jeder CPS-Version (früher Quantum Policy Suite (QPS)) ist '/usr/java/jdk1.7.0_10/bin/', wobei "jdk1.7.0_10" die Version von Java ist und die Version von Java in jedem System unterschiedlich sein kann.

Sie können auch einen Linux-Befehl eingeben, um den genauen Pfad des Jstack-Prozesses zu finden:

```
# find / -iname jstack
```

Jstack wird hier erläutert, um Ihnen die Schritte zur Fehlerbehebung bei Problemen mit hoher CPU-Auslastung aufgrund des Java-Prozesses näher zu bringen. In Fällen mit hoher CPU-Auslastung erfährt man im Allgemeinen, dass ein Java-Prozess die hohe CPU des Systems nutzt.

Vorgehensweise

Schritt 1: Geben Sie den **obersten** Linux-Befehl ein, um zu bestimmen, welcher Prozess eine hohe CPU von der virtuellen Maschine (VM) beansprucht.

```
[root@pcrfclient01 ~]# top
top - 08:36:01 up 221 days, 20:52,  4 users,  load average: 5.86, 3.32, 2.60
Tasks: 1048 total,   1 running, 1037 sleeping,   0 stopped,  10 zombie
Cpu(s): 13.8%us,  4.2%sy,  0.0%ni, 80.0%id,  0.7%wa,  0.2%hi,  1.2%si,  0.0%st
Mem:   5975016k total,  5612888k used,   362128k free,    59776k buffers
Swap:  2097144k total,  1434016k used,   663128k free,   913832k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
14763 root      25   0 10.4g 1.3g 9.8m S  5.9 23.3   5728:23 java
21534 qns       18   0  121m  71m 1460 S  1.7  1.2   6250:45 cisco
 6667 apache    16   0  312m  20m 3984 S  1.3  0.3   0:15.51 httpd
  929 mongod    15   0  572m  97m  71m S  1.0  1.7   1744:19 mongod
14973 root      15   0 13428 2060  940 R  1.0  0.0   0:00.09 top
 4950 apache    16   0  312m  19m 3984 S  0.3  0.3   0:09.06 httpd
11839 apache    16   0  312m  20m 3984 S  0.3  0.3   0:27.41 httpd
12819 apache    16   0  312m  20m 3984 S  0.3  0.3   0:16.89 httpd
    1 root      15   0 10368  628  596 S  0.0  0.0   7:00.45 init
    2 root      RT  -5     0    0    0 S  0.0  0.0   9:12.97 migration/0
```

Aus dieser Ausgabe nehmen Sie die Prozesse heraus, die mehr %CPU verbrauchen. Hier benötigt Java 5,9 %, kann aber mehr CPU verbrauchen, z. B. mehr als 40 %, 100 %, 200 %, 300 %, 400 % usw.

Schritt 2: Wenn ein Java-Prozess eine hohe CPU benötigt, geben Sie einen der folgenden Befehle ein, um herauszufinden, welcher Thread wie viel verbraucht:

```
# ps -C java -L -o pcpu,cpu,nice,state,cputime,pid,tid | sort
```

ODER

```
# ps -C
```

Dieses Display zeigt beispielsweise, dass der Java-Prozess eine hohe CPU (+40 %) sowie die Threads des Java-Prozesses belegt, die für die hohe Auslastung verantwortlich sind.

```
<snip>

0.2 - 0 S 00:17:56 28066 28692
```

```
0.2 - 0 S 00:18:12 28111 28622
0.4 - 0 S 00:25:02 28174 28641
0.4 - 0 S 00:25:23 28111 28621
0.4 - 0 S 00:25:55 28066 28691
 43.9 - 0 R 1-20:24:41 28026 30930
 44.2 - 0 R 1-20:41:12 28026 30927
 44.4 - 0 R 1-20:57:44 28026 30916
 44.7 - 0 R 1-21:14:08 28026 30915
 %CPU CPU NI S TIME     PID  TID
```

Was ist ein Thread?

Angenommen, Sie haben eine Anwendung (d. h. einen einzelnen ausgeführten Prozess) im System. Um jedoch viele Aufgaben ausführen zu können, müssen viele Prozesse erstellt werden, und jeder Prozess erstellt viele Threads. Einige der Threads können Leser, Writer und verschiedene Zwecke sein, z. B. die Erstellung von Call Detail Record (CDR) usw.

Im vorherigen Beispiel verfügt die Java-Prozess-ID (z. B. 28026) über mehrere Threads, darunter 30915, 30916, 30927 und viele mehr.

> **Hinweis**: Die Thread-ID (TID) hat das Dezimalformat.

Schritt 3: Überprüfen Sie die Funktionalität der Java-Threads, die die hohe CPU beanspruchen.

Geben Sie diese Linux-Befehle ein, um die vollständige Jstack-Nachverfolgung abzurufen. Prozess-ID ist die Java-PID, z. B. 28026, wie in der vorherigen Ausgabe gezeigt.

```
# cd  /usr/java/jdk1.7.0_10/bin/

# jstack <process ID>
```

Die Ausgabe des vorherigen Befehls sieht wie folgt aus:

```
2015-02-04 21:12:21
Full thread dump Java HotSpot(TM) 64-Bit Server VM (23.7-b01 mixed mode):

"Attach Listener" daemon prio=10 tid=0x000000000fb42000 nid=0xc8f waiting on
condition [0x0000000000000000]
java.lang.Thread.State: RUNNABLE

"ActiveMQ BrokerService[localhost] Task-4669" daemon prio=10 tid=0x00002aaab41fb800
nid=0xb24 waiting on condition [0x000000004c9ac000]
java.lang.Thread.State: TIMED_WAITING (parking)
at sun.misc.Unsafe.park(Native Method)
- parking to wait for <0x00000000c2c07298>
(a java.util.concurrent.SynchronousQueue$TransferStack)
at java.util.concurrent.locks.LockSupport.parkNanos(LockSupport.java:226)
at java.util.concurrent.SynchronousQueue$TransferStack.awaitFulfill
(SynchronousQueue.java:460)
at java.util.concurrent.SynchronousQueue$TransferStack.transfer
(SynchronousQueue.java:359)
at java.util.concurrent.SynchronousQueue.poll(SynchronousQueue.java:942)
at java.util.concurrent.ThreadPoolExecutor.getTask(ThreadPoolExecutor.java:1068)
at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1130)
at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:615)
at java.lang.Thread.run(Thread.java:722)
```

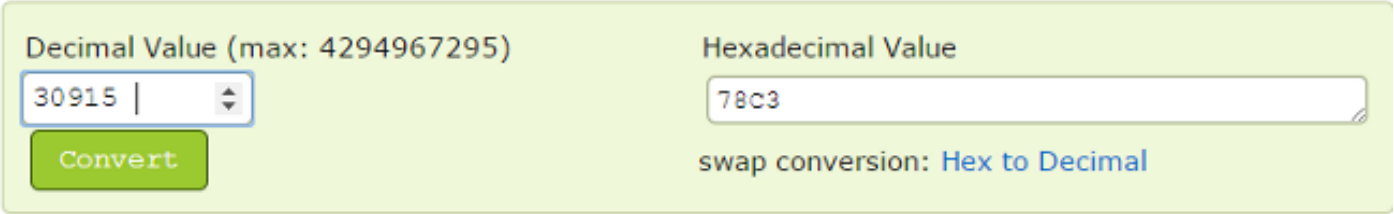
```
"ActiveMQ BrokerService[localhost] Task-4668" daemon prio=10 tid=0x00002aaab4b55800
nid=0xa0f waiting on condition [0x0000000043a1d000]
java.lang.Thread.State: TIMED_WAITING (parking)
at sun.misc.Unsafe.park(Native Method)
- parking to wait for <0x00000000c2c07298>
(a java.util.concurrent.SynchronousQueue$TransferStack)
at java.util.concurrent.locks.LockSupport.parkNanos(LockSupport.java:226)
at java.util.concurrent.SynchronousQueue$TransferStack.awaitFulfill
(SynchronousQueue.java:460)
at java.util.concurrent.SynchronousQueue$TransferStack.transfer
(SynchronousQueue.java:359)
at java.util.concurrent.SynchronousQueue.poll(SynchronousQueue.java:942)
at java.util.concurrent.ThreadPoolExecutor.getTask(ThreadPoolExecutor.java:1068)
at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1130)
at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:615)
at java.lang.Thread.run(Thread.java:722)


<snip>


"pool-84-thread-1" prio=10 tid=0x00002aaac45d8000 nid=0x78c3 runnable
[0x000000004c1a4000]
java.lang.Thread.State: RUNNABLE
at sun.nio.ch.IOUtil.drain(Native Method)
at sun.nio.ch.EPollSelectorImpl.doSelect(EPollSelectorImpl.java:92)
- locked <0x00000000c53717d0> (a java.lang.Object)
at sun.nio.ch.SelectorImpl.lockAndDoSelect(SelectorImpl.java:87)
- locked <0x00000000c53717c0> (a sun.nio.ch.Util$2)
- locked <0x00000000c53717b0> (a java.util.Collections$UnmodifiableSet)
- locked <0x00000000c5371590> (a sun.nio.ch.EPollSelectorImpl)
at sun.nio.ch.SelectorImpl.select(SelectorImpl.java:98)
at zmq.Signaler.wait_event(Signaler.java:135)
at zmq.Mailbox.recv(Mailbox.java:104)
at zmq.SocketBase.process_commands(SocketBase.java:793)
at zmq.SocketBase.send(SocketBase.java:635)
at org.zeromq.ZMQ$Socket.send(ZMQ.java:1205)
at org.zeromq.ZMQ$Socket.send(ZMQ.java:1196)
at com.broadhop.utilities.zmq.concurrent.MessageSender.run(MessageSender.java:146)
at java.util.concurrent.Executors$RunnableAdapter.call(Executors.java:471)
at java.util.concurrent.FutureTask$Sync.innerRun(FutureTask.java:334)
at java.util.concurrent.FutureTask.run(FutureTask.java:166)
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)
```

Nun müssen Sie ermitteln, welcher Thread des Java-Prozesses für die hohe CPU-Auslastung verantwortlich ist.

Betrachten Sie als Beispiel TID 30915, wie in Schritt 2 erwähnt. Sie müssen die TID im Dezimalformat in das Hexadezimalformat konvertieren, da Sie in Jstack trace nur das Hexadezimalformular finden können. Verwenden Sie diesen Konverter, um das Dezimalformat in das Hexadezimalformat zu konvertieren.



Wie Sie in Schritt 3 sehen können, ist die zweite Hälfte der Jstack-Ablaufverfolgung der Thread, der einer der verantwortlichen Threads hinter der hohen CPU-Auslastung ist. Wenn Sie den 78C3 (Hexadezimalformat) im Jstack-Trace finden, finden Sie diesen Thread nur als 'nid=0x78c3'. Daher finden Sie alle Threads dieses Java-Prozesses, die für einen hohen CPU-Verbrauch verantwortlich sind.

> Hinweis: Sie müssen sich vorerst nicht auf den Zustand des Threads konzentrieren. Als interessanter Punkt wurden einige Zustände der Threads wie Runnable, Blocked, Timed_Waiting und Waiting gesehen.

Alle vorherigen Informationen helfen CPS und anderen Technologieentwicklern Ihnen dabei, die Ursache für das Problem der hohen CPU-Auslastung im System/VM zu ermitteln. Erfassen Sie die zuvor genannten Informationen, wenn das Problem auftritt. Wenn die CPU-Auslastung wieder auf den Normalzustand zurückgesetzt ist, können die Threads, die das hohe CPU-Problem verursacht haben, nicht mehr bestimmt werden.

CPS-Protokolle müssen ebenfalls erfasst werden. Nachfolgend finden Sie die Liste der CPS-Protokolle der VM 'PCRFclient01' unter dem Pfad '/var/log/brehop':

- konsolidierter Motor
- konsolidierte qns

Ermitteln Sie außerdem die Ausgabe dieser Skripts und Befehle vom PCRFclient01 VM:

- # diagnostics.sh (Dieses Skript wird möglicherweise nicht auf älteren Versionen von CPS ausgeführt, z. B. QNS 5.1 und QNS 5.2.)
- # df -kh
- # Top