Gebruik van hoge CPU's voor probleemoplossing in Java

Inhoud

[Inleiding](#)
[Probleemoplossing met Jstack](#)
[Wat is Jstack?](#)
[Waarom heb je Jstack nodig?](#)
[Procedure](#)
[Wat is een draad?](#)

Inleiding

Dit document beschrijft Java Stack (Jstack) en hoe u deze kunt gebruiken om de grondoorzaak van het gebruik van hoge CPU's in Cisco Policy Suite (CPS) te bepalen.

Probleemoplossing met Jstack

Wat is Jstack?

Jstack maakt een geheugen-stort van een actief Java-proces (in CPS is QNS een Java-proces). Jstack heeft alle details van dat Java-proces, zoals threads/toepassingen en de functionaliteit van elke thread.

Waarom heb je Jstack nodig?

Jstack biedt het Jstack-spoor zodat engineers en ontwikkelaars de status van elke thread kunnen leren kennen.

De Linux-opdracht die wordt gebruikt om het Jstack-spoor van het Java-proces te verkrijgen, is:

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

De locatie van het Jstack-proces in elke CPS (voorheen bekend als Quantum Policy Suite (QPS)) versie is '/usr/java/jdk1.7.0_10/bin/' waar 'jdk1.7.0_10' de versie van Java is en de versie van Java in elk systeem kan verschillen.

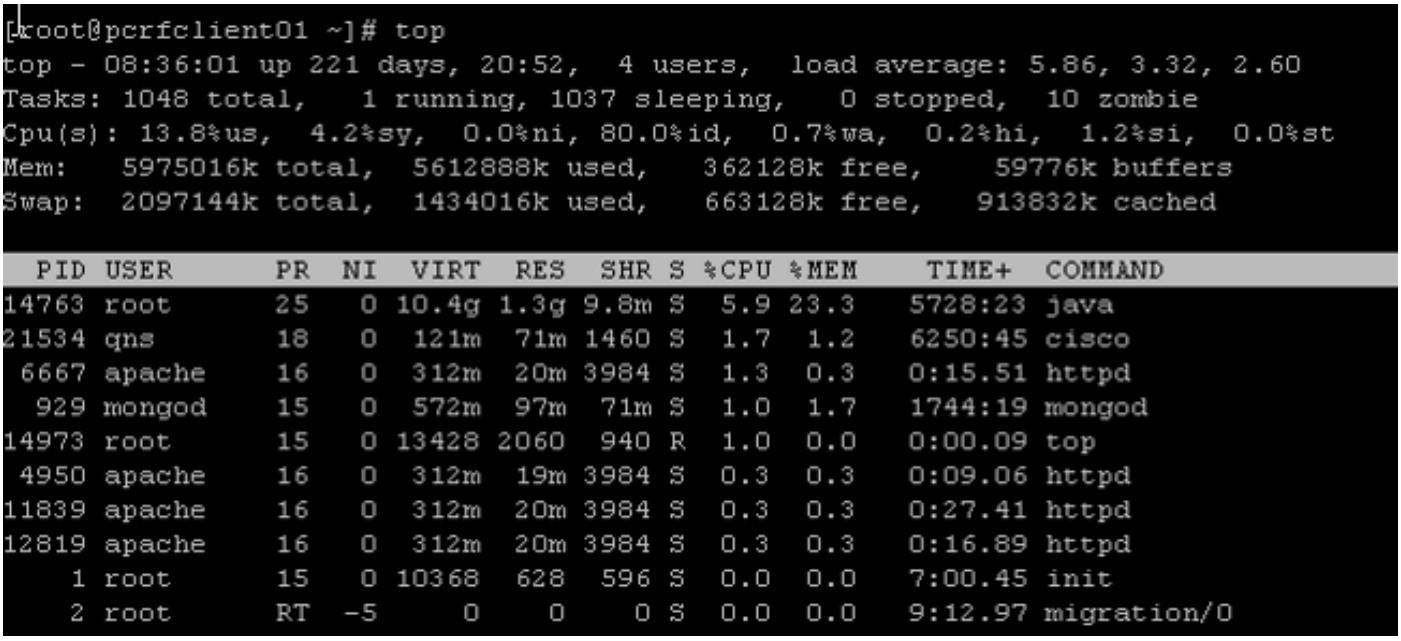
U kunt ook een Linux-opdracht invoeren om de exacte route van het Jstack-proces te vinden:

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

Jstack wordt hier uitgelegd om u bekend te maken met de stappen om problemen met het hoge CPU-gebruik op te lossen vanwege het Java-proces. Bij gebruik met hoge CPU's leert u over het algemeen dat een Java-proces gebruik maakt van de hoge CPU's van het systeem.

Procedure

Stap 1: Voer de opdracht top Linux in om te bepalen welk proces hoge CPU van de virtuele machine (VM) verwerkt.



Haal uit deze uitvoer de processen die meer %CPU's verbruiken. Java neemt 5,9 % in beslag, maar kan meer CPU's gebruiken zoals meer dan 40%, 100%, 200%, 300%, 400%, enzovoort.

Stap 2: Als een Java-proces hoge CPU's gebruikt, voert u een van deze opdrachten in om te weten te komen welke thread meer verbruikt dan:

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

OF

```
# ps -C
```

Deze weergave laat bijvoorbeeld zien dat het Java-proces een hoge CPU (+40%) en een hoog Java-proces vergt, evenals de verbindingen van het Java-proces dat voor een hoog gebruik verantwoordelijk is.

```
<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
```

Wat is een draad?

Stel dat u een toepassing (dat wil zeggen één lopend proces) binnen het systeem hebt. Om veel taken uit te voeren, moet u echter veel processen creëren en elk proces creëert vele draden. Sommige threads zouden kunnen bestaan uit het maken van lezers, schrijvers en andere doeleinden, zoals Call Detail Record (CDR), enzovoort.

In het vorige voorbeeld heeft de Java-procesID (bijvoorbeeld 28026) meerdere threads, waaronder 30915, 30916, 30927 en nog veel meer.

> Opmerking: De thread ID (TID) is in decimale notatie.

Stap 3: Controleer de functionaliteit van de Java-draden die de hoge CPU gebruiken.

Voer deze Linux-opdrachten in om het volledige Jstack-spoor te verkrijgen. ProcesID is de Java-PID, bijvoorbeeld 28026 zoals in de vorige uitvoer wordt weergegeven.

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

# jstack <process ID>
```

Uitvoer van de vorige opdracht ziet er zo uit:

```
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)
```

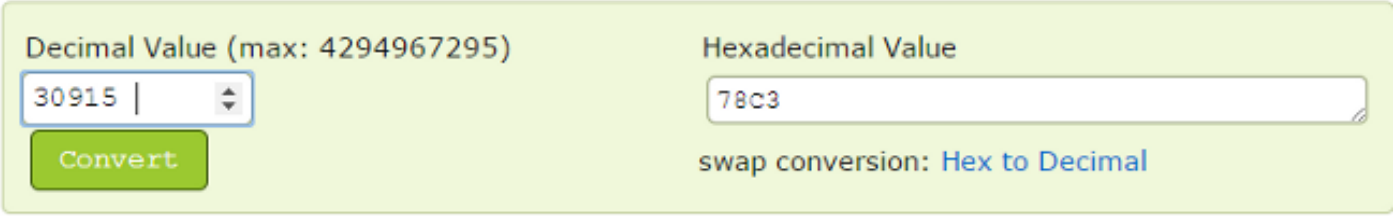
```
"ActiveMQ BrokerService[localhost] Task-4668" daemon prio=10 tid=0x00002aaab4b55800
nid=0xa0f waiting on condition [0x0000000043a1d000]
java.lang.Thread.State: TIMED_WAITING (parking)
at sun.misc.Unsafe.park(Native Method)
- parking to wait for <0x00000000c2c07298>
(a java.util.concurrent.SynchronousQueue$TransferStack)
at java.util.concurrent.locks.LockSupport.parkNanos(LockSupport.java:226)
at java.util.concurrent.SynchronousQueue$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)
```

U moet nu bepalen welke thread van het Java-proces verantwoordelijk is voor het hoge CPU-gebruik.

Als voorbeeld, kijk naar TID 30915 zoals vermeld in Stap 2. U moet de TID in decimaal naar hexadecimaal formaat converteren omdat in Jstack-sporen alleen het hexadecimale formulier kunnen vinden. Gebruik deze converter om het decimale formaat om te zetten in het hexadecimale formaat.



Zoals u in Stap 3 kunt zien, is de tweede helft van het spoor van Jstack de draad die achter het hoge CPU-gebruik ligt. Wanneer u 78C3 (hexadecimale formaat) in de lijn van Jstack vindt, dan zal u deze draad slechts als 'nid=0x78c3' vinden. Daarom kunt u alle draden van dat Java-proces vinden die verantwoordelijk zijn voor een hoog CPU-verbruik.

> Opmerking: U hoeft zich voorlopig niet te richten op de status van de thread. Als een belangrijk punt zijn er enkele toestanden te zien van de draden zoals Runnable, Blocked, TTime_Waiting en Waiting.

Met al de vorige informatie helpen CPS en andere technologieontwikkelaars u bij het bereiken van de basisoorzaak van de hoge CPU-gebruikskwestie in het systeem/VM. Leg de eerder genoemde informatie op het moment dat het probleem wordt weergegeven. Zodra het CPU-gebruik weer normaal is, kunnen de draden die de hoge CPU-kwestie hebben veroorzaakt, niet meer worden bepaald.

CPS-bestanden moeten ook worden opgenomen. Hier is de lijst van CPS-logbestanden van de "PCRFclient01" VM onder het pad"/var/log/Broadhop":

- geconsolideerde motor
- geconsolideerd

Verkrijg ook uitvoer van deze scripts en opdrachten van de PCRFclient01 VM:

- # diagnostiek.sh (Dit script kan mogelijk niet worden uitgevoerd op oudere versies van CPS, zoals QNS 5.1 en QNS 5.2)
- # df-kh
- # boven