How to solve java.lang.OutOfMemoryError: unable to create new native thread

In Java you can stumble upon two kind of **Out of Memory** errors:

- The ***java.lang.OutOfMemoryError Java heap space error*** : This exception will be triggered when the application attempts to allocate more data into the heap space area, but there is not enough room for it. Although there might be plenty of memory available on your machine, you have hit the maximum amount of memory allowed by your JVM, which can be set through the -Xmx parameter
- The ***java.lang.OutOfMemoryError: Unable to create new native thread*** happens whenever the JVM asks for a new thread from the OS. If the underlying OS cannot allocate a new native thread, this OutOfMemoryError will be thrown.

1) Check Threads system wide settings

The **/proc/sys/kernel/threads-max** file provides a system-wide limit for the number of threads. The root user can change that value if they wish to:

```
$ echo 100000 > /proc/sys/kernel/threads-max
```

You can check the current number of running threads through the /proc/loadavg filesystem:

```
$ cat /proc/loadavg
0.41 0.45 0.57 3/749 28174
```

> Watch the **fourth** field! This field consists of two numbers separated by a slash (/). The first of these is the number of currently executing kernel scheduling entities (processes, threads); this will be less than or equal to the number of CPUs. The value after the slash is the number of kernel scheduling entities that currently exist on the system. In this case you are running 749 threads

2) Check number of processes per user

On a Linux box, threads are essentially just **processes with a shared address space.** Therefore, you have to check if your OS allows you enough processes for user. This can be checked through:

```
ulimit -a
core file size          (blocks, -c) 0
data seg size           (kbytes, -d) unlimited
scheduling priority             (-e) 0
file size               (blocks, -f) unlimited
pending signals                 (-i) 515005
max locked memory       (kbytes, -l) 64
max memory size         (kbytes, -m) unlimited
open files                      (-n) 4096
pipe size            (512 bytes, -p) 8
POSIX message queues     (bytes, -q) 819200
real-time priority              (-r) 0
stack size              (kbytes, -s) 10240
cpu time               (seconds, -t) unlimited
max user processes              (-u) 1024
virtual memory          (kbytes, -v) unlimited
file locks                      (-x) unlimited
```

The default number of process per users is 1024 by default. At this point we will **count the number of processes running**. The number of processes running can be counted with a ps output:

```
$ ps -elf | wc -l
220
```

This number however does not consider the threads which can be spawned by a process. If you try to run ps with a -T you will see all of the threads as well:

```
$ ps -elfT | wc -l
385
```

As you can see the process count increased significantly due to threads. Normally this is never any type of problem, However in Java based applications this can cause your system to run into system limits! Let's continue our investigation. Let's see how many Threads are spawned by your JBoss Process. You can do it in at least two ways:

```
$ ps -p JBOSSPID -lfT | wc -l
```

The above shell will return the number of **Lightweight Processes** created for a Process indicated by the PID. This should match with the **Thread Dump** count generated by jstack:

```
$ jstack -l JBOSSPID | grep tid | wc -l
```

Now you should have evidence or not that you need to increase the number of processes for the user. This can be done with the following command:

```
$ ulimit -u 4096
```

3) Check your threads PID limit

Once that you have counted the number of threads, then you should verify that you are not hitting system limits, specified by the kernel.pid_max limit parameter. You can check this value by executing:

```
$ sysctl -a | grep kernel.pid_max

kernel.pid_max = 32768
```

4) Reduce the Thread Stack size

Another option which you can use, if you are not able to modify the OS settings is **reducing the stack size**. The JVM has an interesting implementation, by which the more memory is allocated for the heap (not necessarily used by the heap), the less memory available in the stack, and since threads are made from the stack, in practice this means more "memory" in the heap sense (which is usually what people talk about) results in less threads being able to run concurrently.

First of all check the default Thread Stack size which is dependent on your Operating System:

```
$  java -XX:+PrintFlagsFinal -version | grep ThreadStackSize
     intx ThreadStackSize                          = 1024                                {pd product}
```

As you can see, the default Thread Stack Size is 1024 kb in our machine. In order to reduce the stack size, add "**-Xss**" option to the JVM options. In JBoss EAP 6 / WildFly the minimum Thread stack size is 228kb. You can change it in Standalone mode by varying the JAVA_OPTS as in the following example:

```
JAVA_OPTS="-Xms128m -Xmx1303m -Xss256k"
```

In Domain Mode, you can configure the jvm element at various level (Host, Server Group, Server). There you can set the requested Stack Size as in the following section:

```
<jvm name="default">
            <heap size="64m" max-size="256m"/>
            <jvm-options>
                <option value="-server"/>
                <option value="-Xss256k"/>
            </jvm-options>
</jvm>
```