The JVM and containers: a perfect match?

richard.swart-at-nlighten.nl

We will look at

Resources:

- Memory
- CPU

Using my laptop as reference:

- 32 GB
- 4 cores

Software:

- OpenJDK 11.0.5
- Docker 17.05.0-ce
- K3S v1.17.0+k3s.1 (Containerd)

Test application

```
public class SystemInfo {

  public static void main(String[] args) {
    long heapSize = Runtime.getRuntime().totalMemory();
    long heapMaxSize = Runtime.getRuntime().maxMemory();
    long heapFreeSize = Runtime.getRuntime().freeMemory();

    System.out.println("cur heapsize         : " + formatSize(heapSi
    System.out.println("max heapsize         : " + formatSize(heapMa
    System.out.println("free heapsize        : " + formatSize(heapFr
    System.out.println("available processors : " + Runtime.getRuntim
  }


  public static String formatSize(long v) {
    if (v < 1024) return v + " B";
    int z = (63 - Long.numberOfLeadingZeros(v)) / 10;
    return String.format("%.1f %sB", (double)v / (1L << (z*10)), " K
  }
}
```

Directly on host I

We usually start a java program like this:

```
$ java -Xms256m -Xmx256m SystemInfo
```

Output:

```
cur heapsize          : 256.0 MB
max heapsize          : 256.0 MB
free heapsize         : 254.7 MB
available processors : 4
```

What would happen if we don't pass heap size params?

Directly on host II

Let's try:

```
$ java SystemInfo
```

Output:

```
cur heapsize          : 504.0 MB
max heapsize          : 7.8 GB
free heapsize         : 501.9 MB
available processors : 4
```

Since we did not pass any heap size params the JVM decided for us. How?

Directly on host III

Check JVM flags

```
java -XX:+PrintFlagsFinal SystemInfo \
| grep -E " InitialRAMPercentage | MaxRAMPercentage | MinRAMPercenta
```

Output:

```
   double InitialRAMPercentage                     = 1.562500
   double MaxRAMPercentage                         = 25.000000
   double MinRAMPercentage                         = 50.000000
```

- InitialRAMPercentage is used to calculate initial heap size when InitialHeapSize / -Xms is not set.
- For systems with small physical memory MaxHeapSize is set at:

  ```
  phys_mem * MinRAMPercentage / 100  (if < 96M)
  ```

- Otherwise MaxHeapSize is aet at:

  ```
  MAX(phys_mem * MaxRAMPercentage / 100, 96M)
  ```

With explicit heap size:

```
docker run --rm -e "JAVA_OPTS=-Xms256m -Xmx256m" localhost:5000/sys-
```

Output:

```
cur heapsize          : 256.0 MB
max heapsize          : 256.0 MB
free heapsize         : 254.7 MB
available processors : 4
```

Identical as on the host.

Docker II

No heap size params:

```
docker run --rm localhost:5000/sys-info
```

Output:

```
cur heapsize          : 500.0 MB
max heapsize          : 7.8 GB
free heapsize         : 498.1 MB
available processors : 4
```

Identical as on the host.

Kubernetes I

With explicit heap size:

```
kubectl run sysinfo-container --generator=run-pod/v1 --image=localho
```

Output:

```
cur heapsize         : 247.5 MB
max heapsize         : 247.5 MB
free heapsize        : 246.1 MB
available processors : 1
```

So on K8S I get:

- slightly less memory allocated
- but only 1 cpu

???????

Kubernetes II

No heap size param:

```
kubectl run sysinfo-container --generator=run-pod/v1 --image=localho
```

Output:

```
cur heapsize          : 483.4 MB
max heapsize          : 7.6 GB
free heapsize         : 475.4 MB
available processors : 1
```

Same pattern :

- 'slightly' less memory
- only 1 cpu

Kubernetes III

Let's add some debug logging:

```
kubectl run sysinfo-container --generator=run-pod/v1 --image=localho
```

Output:

```
[0.139s][trace][os,container] Path to /cpu.cfs_quota_us is /sys/fs/c
[0.139s][trace][os,container] CPU Quota is: -1
[0.139s][trace][os,container] Path to /cpu.cfs_period_us is /sys/fs/
[0.139s][trace][os,container] CPU Period is: 100000
[0.139s][trace][os,container] Path to /cpu.shares is /sys/fs/cgroup/
[0.139s][trace][os,container] CPU Shares is: 2
[0.139s][trace][os,container] CPU Share count based on shares: 1
[0.139s][trace][os,container] OSContainer::active_processor_count: 1
[0.142s][trace][os,container] Path to /memory.limit_in_bytes is /sys
[0.142s][trace][os,container] Memory Limit is: 9223372036854771712
[0.142s][trace][os,container] Memory Limit is: Unlimited
[0.142s][debug][os,container] container memory limit unlimited: -1,
```

Ok, so it is using cgroup information. But why the difference with a plain Docker container?

CPU limiting inside a container

There are three ways a Linux container can have its CPU limited:

- **cpu_shares**. A relative figure that gives the container a share of the CPU.
- **cpu_limit**. A hard CPU time that can be used per cpu_period.
- **cpu_sets**. Run the container on specific CPU's.

By default:

- Docker sets no limit and CPU shares to 1024
- K8S sets no limit and CPU shares to 2

How the JVM counts CPU's inside a container

From JDK source code:

```
* Algorithm:
 *
 * If user specified a quota (quota != -1), calculate the number of
 * required CPUs by dividing quota by period.
 *
 * If shares are in effect (shares != -1), calculate the number
 * of CPUs required for the shares by dividing the share value
 * by PER_CPU_SHARES.
 *
 * All results of division are rounded up to the next whole number.
 *
 * If neither shares or quotas have been specified, return the
 * number of active processors in the system.
 *
 * If both shares and quotas have been specified, the results are
 * based on the flag PreferContainerQuotaForCPUCount.  If true,
 * return the quota value.  If false return the smallest value
 * between shares or quotas.
 *
 * If shares and/or quotas have been specified, the resulting number
 * returned will never exceed the number of active processors.
```

But there is a snag!

Again from JDK source code:

```
/* cpu_shares
 *
 * Return the amount of cpu shares available to the process
 *
 * return:
 *    Share number (typically a number relative to 1024)
 *                 (2048 typically expresses 2 CPUs worth of process
 *    -1 for no share setup
 *    OSCONTAINER_ERROR for not supported
 */
int OSContainer::cpu_shares() {
  GET_CONTAINER_INFO(int, cpu, "/cpu.shares",
                     "CPU Shares is: %d", "%d", shares);
  // Convert 1024 to no shares setup
  if (shares == 1024) return -1;

  return shares;
}
```

They made the Docker default of 1024 cpu_shares equivalent to unlimited!

Leading to weird behavior

The following gives 'all' cpu's:

```
docker run --rm --cpu-shares=1024 sys-info
```

```
cur heapsize         : 500.0 MB
max heapsize         : 7.8 GB
free heapsize        : 498.1 MB
available processors : 4
```

But doubling CPU shares only gives me 2!

```
docker run --rm --cpu-shares=2048 sys-info
```

```
cur heapsize         : 500.0 MB
max heapsize         : 7.8 GB
free heapsize        : 498.1 MB
available processors : 2
```

Similar on K8S

The following gives 'all' cpu's:

```
kubectl run sysinfo-container --generator=run-pod/v1 --image=localho
```

```
cur heapsize         : 500.0 MB
max heapsize         : 7.8 GB
free heapsize        : 498.1 MB
available processors : 4
```

But doubling CPU requests only gives me 2.

```
kubectl run sysinfo-container --generator=run-pod/v1 --image=localho
```

```
cur heapsize         : 500.0 MB
max heapsize         : 7.8 GB
free heapsize        : 498.1 MB
available processors : 2
```

What if I use limits on K8S

Limiting the CPU to max 1 gives:

```
kubectl run sysinfo-container --generator=run-pod/v1 --image=localho
```

```
cur heapsize         : 483.4 MB
max heapsize         : 7.6 GB
free heapsize        : 475.4 MB
available processors : 1
```

Limiting the CPU to max 2 gives:

```
kubectl run sysinfo-container --generator=run-pod/v1 --image=localho
```

```
cur heapsize         : 500.0 MB
max heapsize         : 7.8 GB
free heapsize        : 498.1 MB
available processors : 2
```

So `--limits` gives a predictable number of CPUs but ..... you probably don't want to set them.

Limits problems

1. CPU CFS quota often leads to latency.
2. Memory limits can lead to OOMKill behavior

and specifically on K8S:

1. Less density due to limits >> requests

Quality of Service (QoS) on K8S

Three types of QoS for pods:

- **Guaranteed**: all containers of the pod have limits==requests
- **Burstable**: some containers of the pod have limits > requests
- **BestEffort**: none of the containers have request/limits set

```
kubectl describe pod sysinfo-container
```

```
.....
    Limits:
      cpu:     2
      memory:  256Mi
    Requests:
      cpu:        2
      memory:     128Mi
.....
QoS Class:       Burstable
```

Overcommit on K8S

Overcommit: Limits > Request (Burstable)

For CPU: fine, completely fair scheduling

For memory: fine, as long as demand < node capacity

If the latter condition is not met you may see unpredictable OOMKiller action.

Rule of thumb for running a JVM on K8S?

For a generic K8S cluster targeted at good density:

Memory:

- Don't overcommit memory, so `requests.memory=limits.memory`
- Set max heap size equal to half container memory, so `Xms=Xmx=1/2*requests.memory`, or probably better:
- Don´t set max heap but set `-XX:MaxRAMPercentage=50.0 -XX:InitialRAMPercentage=50.0`

CPU:

- Disable CFS quota (`--cpu-cfs-quota=false`) if you can
- Don't set CPU limits
- Set `-XX:ActiveProcessorCount=xx` for each JVM to a sensible number (e.g. 2)

Cloud Foundry Java Buildpack Memory Calculator

Go executable that calculates memory settings based on:

- Total available memory
- Estimated loaded class count (35% of classes found on classpath)
- Thread count (stack)
- Head room

Can be used in an entrypoint script to calculate your memory settings:

```
docker run -it --rm  -m 1G memcalc
INFO - input for memory calculator: -head-room=10 --jvm-options='' -
INFO - resulting MEMORY_OPTS: -XX:MaxDirectMemorySize=10M -XX:MaxMet
```

.....