PITFALLS ABOUT RUNNING JAVA IN A CONTAINER

DISCLOSURE

- The slides are copied from different sources from internet to make it easy to discuss the pitfalls of running JAVA in a containers.

- The references are provided at the end of the slides for the sources from which the slides are copied.

- Most of the slides are inspired from reference [1]. Where the slides does not inspire from the content available in [1], the reference is provided in the slide title.

SUMMARY – TL;DR

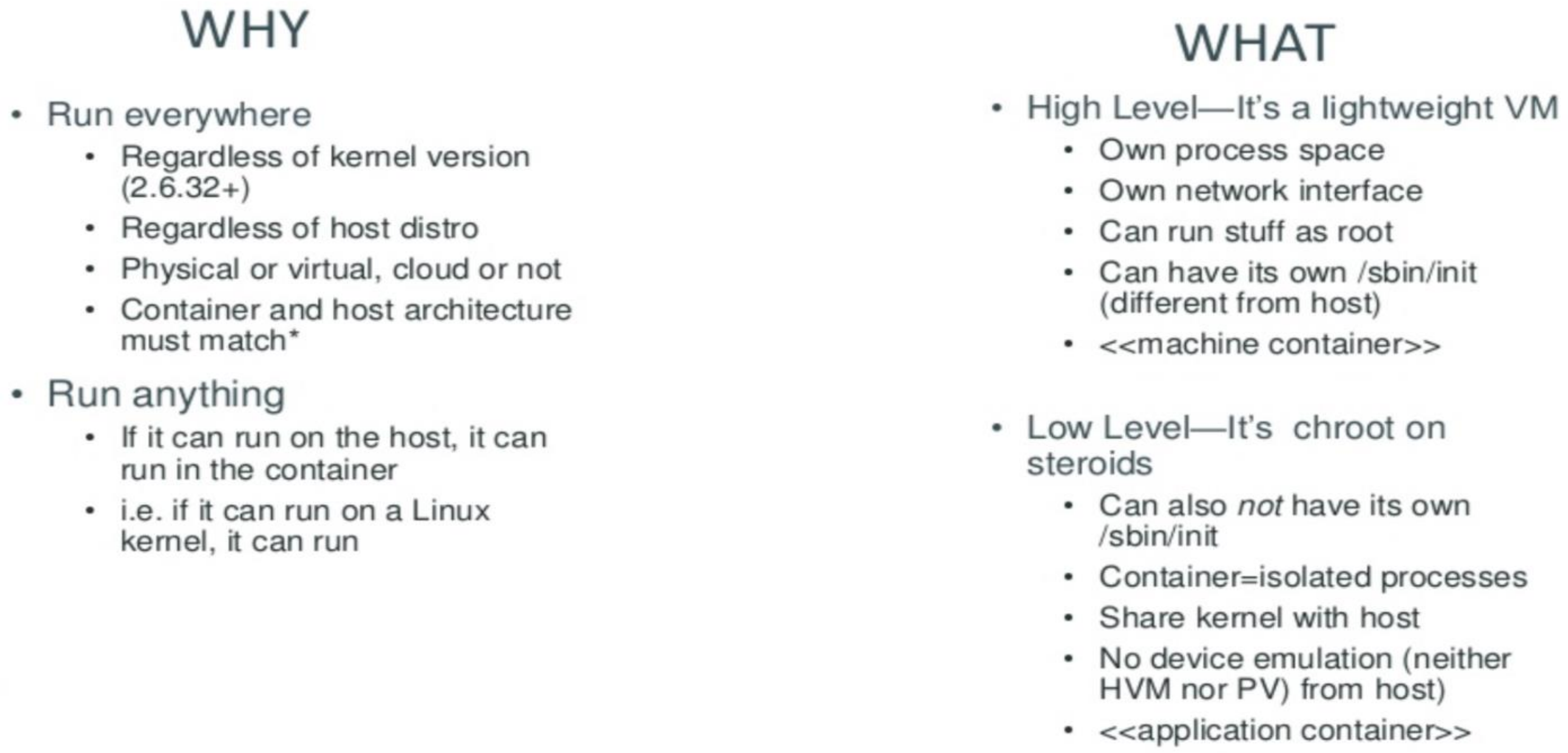
- The Java Virtual Machine (not even with the Java 9 release) is not fully aware of the isolation mechanisms containers use internally, this can lead to unexpected behavior between different environments (e.g., test vs production).

- To avoid this behavior one should consider overriding some default parameters (which are usually set by the memory and processors available on the node) to match the container limits.

TABLE OF CONTENTS

CONTAINERS

CONTAINERS (ADAPTED FROM [3])



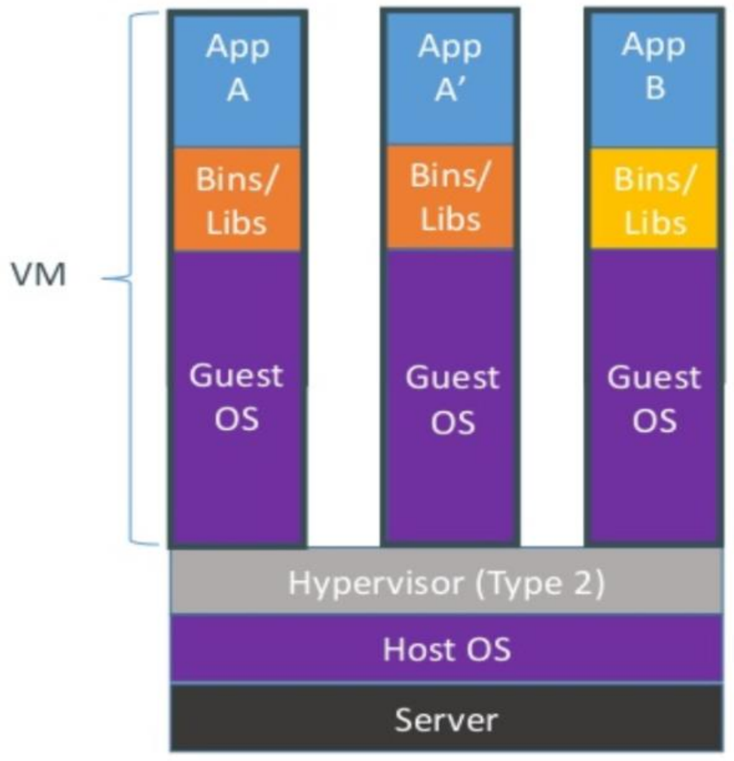
WHY

- Run everywhere
  - Regardless of kernel version (2.6.32+)
  - Regardless of host distro
  - Physical or virtual, cloud or not
  - Container and host architecture must match*
- Run anything
  - If it can run on the host, it can run in the container
  - i.e. if it can run on a Linux kernel, it can run

WHAT

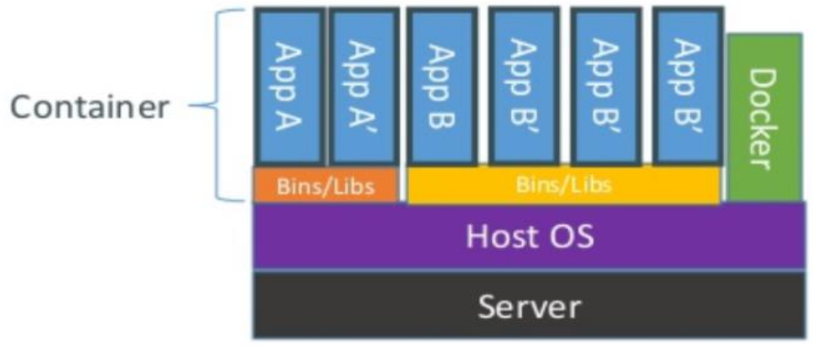
- High Level—It's a lightweight VM
  - Own process space
  - Own network interface
  - Can run stuff as root
  - Can have its own /sbin/init (different from host)
  - <<machine container>>
- Low Level—It's chroot on steroids
  - Can also *not* have its own /sbin/init
  - Container=isolated processes
  - Share kernel with host
  - No device emulation (neither HVM nor PV) from host)
  - <<application container>>

CONTAINERS VS VMS (ADAPTED FROM [3])



Containers are isolated, but share OS and, where appropriate, bins/libraries

...result is significantly faster deployment, much less overhead, easier migration, faster restart



CONTAINERS VS VMS (CONTD…)

- Containers are basically just isolated linux process groups.
- This means that all "containers" running on the same host, are process groups running on the same linux kernel of the host
- Let us understand this by actually running some docker commands:
  - docker run ubuntu sleep 1000 &
  - docker run ubuntu sleep 5000 &
  - docker ps
  - ps faux
- Containers provides process isolation using two linux features – namespaces and cgroups.

NAMESPACES

ISOLATION

NAMESPACES

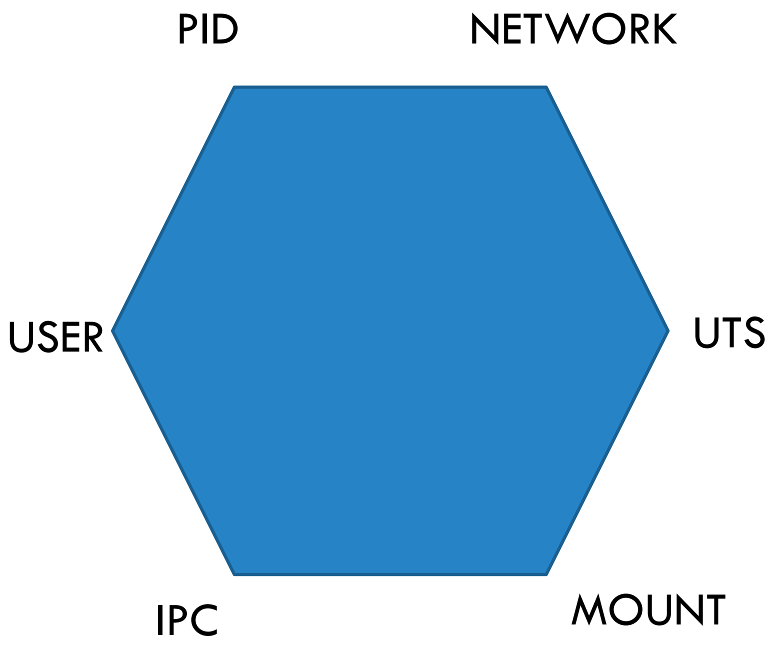
- Namespaces are basically used for providing different isolated views on the system.

- So each container can see its own view on different namespaces like process IDs, on network IDs, or user IDs. It works the same for processes. So for example, in different containers the process ID 1.

- Six different namespaces are available - PID (processes), Network (network interfaces, routing, ….), UTS (hostname), Mount (mount point, file system), IPC (System V IPC), User (UIDs)

Let us see the process view from inside the containers.

- docker ps
- docker exec –it <container id>
- ps faux



Namespaces Hexagon

NAMESPACES – BLINDER’S ANALOGY

- Namespaces can be seen as blinders which are put on horses to restrict their view.

- Similar to the blinders, namespaces make sure that the process only see the namespace in which they are created.

- If everyone put a container around self in this conference room, they will feel that they are probably the only one in this conference room.

CGROUPS

RESTRICTION

CGROUPS

- control groups can be used for both limiting access and also for accounting purposes.

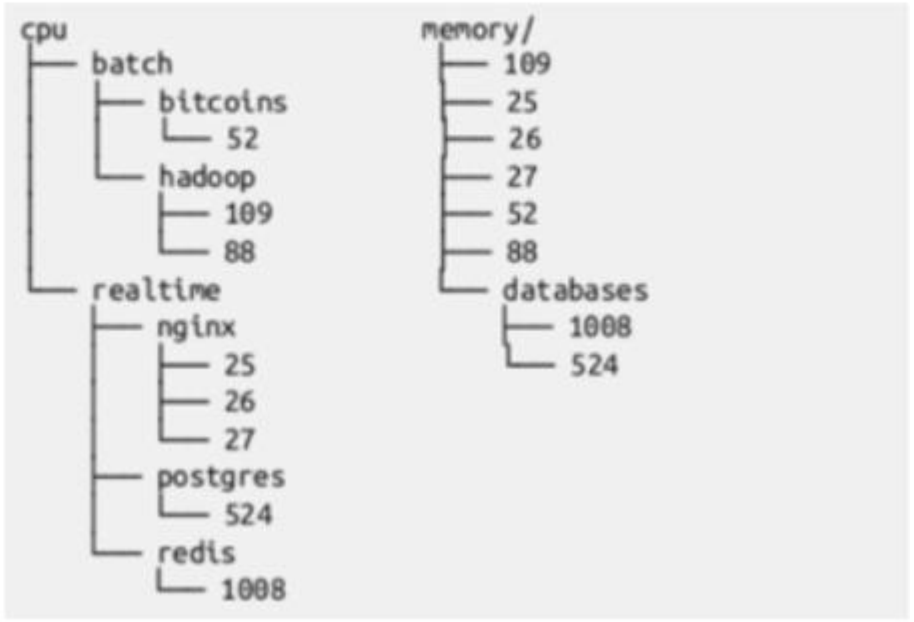
- As everything in Linux or Unix, it's just like a hierarchical folder which can be viewed as a tree.

- Each sub-system (CPU, memory) has a hierarchy tree

- Each process belongs to exactly one node in each hierarchy

- Each hierarchy starts with one node

- Each node = group of processes (sharing the same resources)



```
cpu                              memory/
├── batch                         ├── 109
│   ├── bitcoins                  ├── 25
│   │   └── 52                    ├── 26
│   └── hadoop                    ├── 27
│       ├── 109                   ├── 52
│       └── 88                    ├── 88
└── realtime                      └── databases
    ├── nginx                         ├── 1008
    │   ├── 25                        └── 524
    │   ├── 26
    │   └── 27
    ├── postgres
    │   └── 524
    └── redis
        └── 1008
```

CGROUPS – MEMORY RESTRICTION

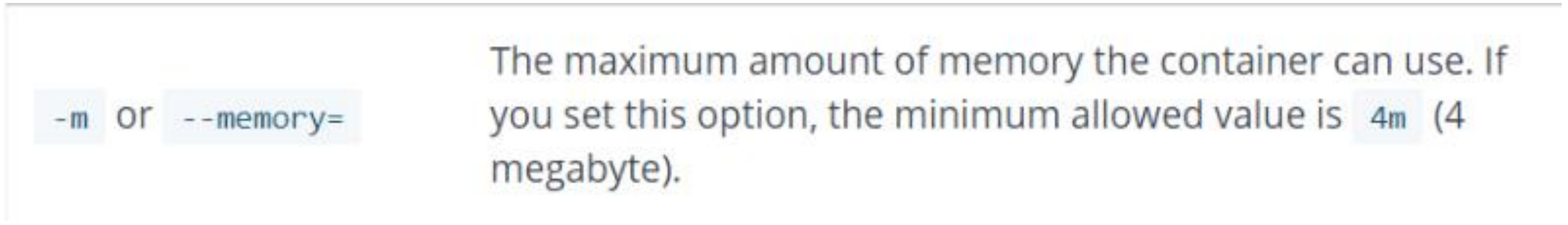
- When using docker, we can set a hard limit of 40MB for our containers as follows

- `docker run –it –m 40m java8`

- Hard limits will trigger a per-group OOM killer.

- This often requires some changed mindset for Java developers, as they are used to a `OutOfMemoryError` which they can react to accordingly.

- But in case of the containers with a hard memory limit, the entire container will simply be killed without warning.



`-m` or `--memory=`	The maximum amount of memory the container can use. If you set this option, the minimum allowed value is `4m` (4 megabyte).

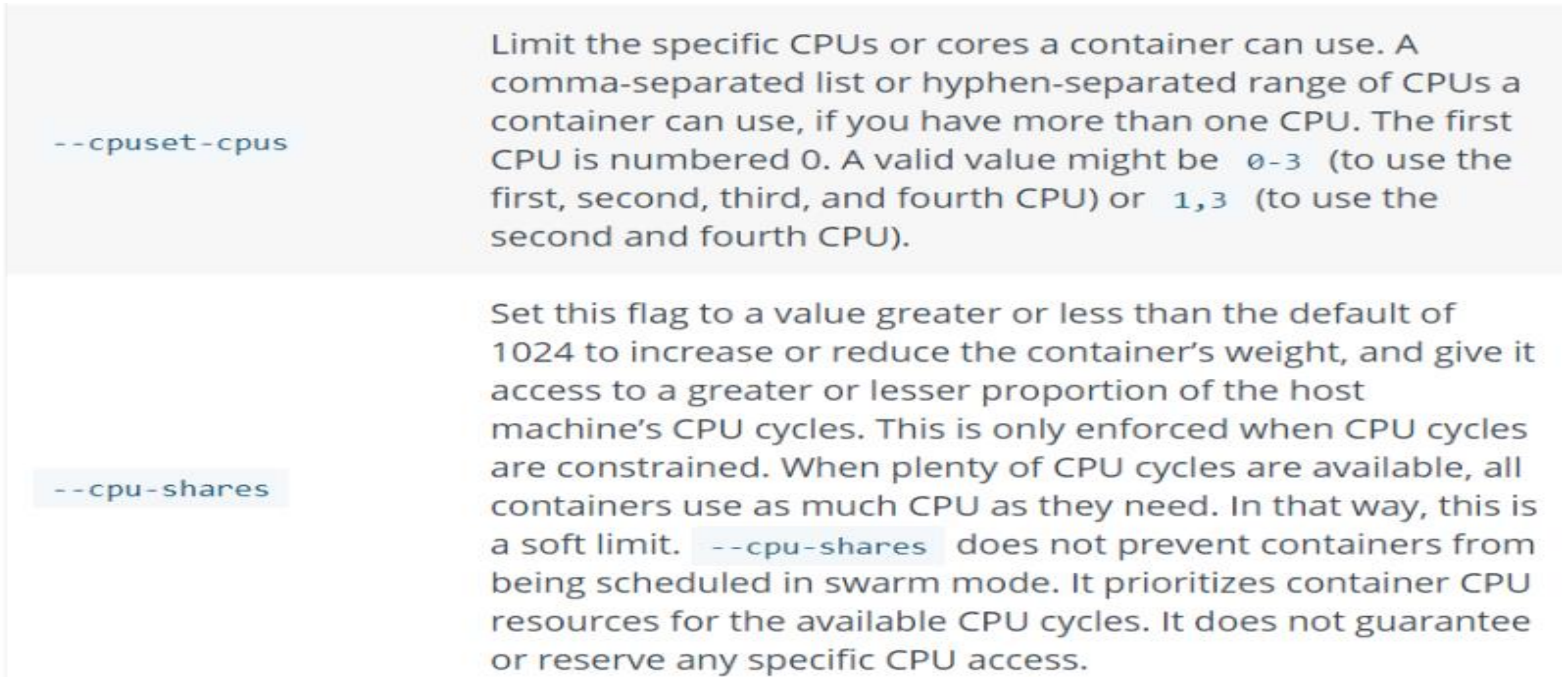
CGROUPS – OOM – KILLER?? (COPIED FROM [11])

Save us, OOM Killer, Java is eating up all our resources.
Let me handle it.
SIGKILL
Take this sigkill, you hungry beast!
POW
KILL
KILL-9
SIGKILL
It didn't work! We're all gonna die!
Wake up, damn sysadmin! Reboot the machine!
java.hungry(); you.die();

Starting Linux...
I don't know why, but I'm not very fond of this kid.
Don't be annoying. The kid is cool.
I'm hungry

CGROUPS – CPU ISOLATION

- There are two choices for CPU isolation – CPU shares and CPU-sets. Corresponding docker options:

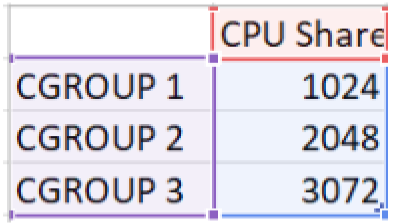


`--cpuset-cpus`	Limit the specific CPUs or cores a container can use. A comma-separated list or hyphen-separated range of CPUs a container can use, if you have more than one CPU. The first CPU is numbered 0. A valid value might be `0-3` (to use the first, second, third, and fourth CPU) or `1,3` (to use the second and fourth CPU).
`--cpu-shares`	Set this flag to a value greater or less than the default of 1024 to increase or reduce the container's weight, and give it access to a greater or lesser proportion of the host machine's CPU cycles. This is only enforced when CPU cycles are constrained. When plenty of CPU cycles are available, all containers use as much CPU as they need. In that way, this is a soft limit. `--cpu-shares` does not prevent containers from being scheduled in swarm mode. It prioritizes container CPU resources for the available CPU cycles. It does not guarantee or reserve any specific CPU access.

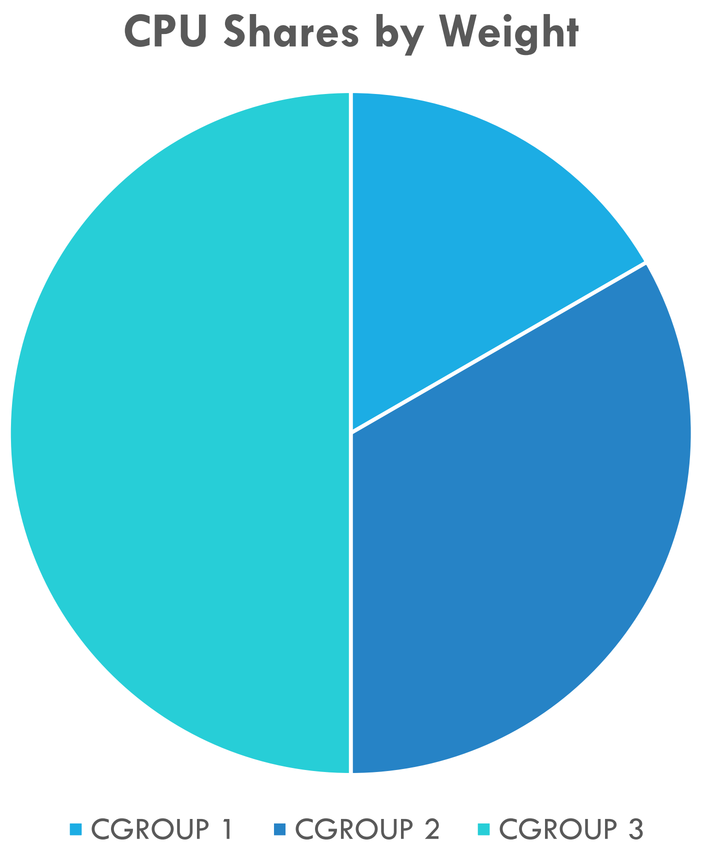
CGROUPS – CPU SHARES RESTRICTION

- CPU shares are the default CPU isolation and basically provide a priority weighting across all all cpu cycles across all cores.

- The default weight of any process is 1024, so if start a container as follows `docker run –it –c 512 stress`, it will receive less CPU cycles than a default process/containers



	CPU Share
CGROUP 1	1024
CGROUP 2	2048
CGROUP 3	3072



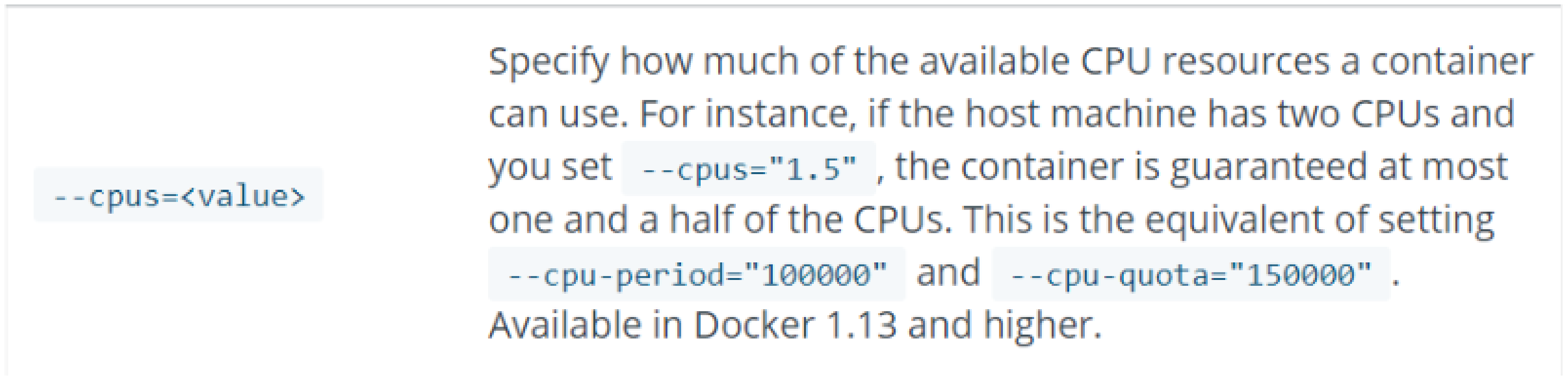
CGROUPS – CPUSET ISOLATION

- CPU sets are slightly different. They allow to limit container to specific CPU(s).

- This is mostly used to avoid processes bouncing between CPUs, but is also relevant for NUMA systems where different CPU have fast access to different memory regions (and hence you want your container to only utilize the CPU with fast access to the same memory region).

DOCKER – CPUS OPTIONS

- Starting with docker 1.13, docker provides a –cpus options which is seen to be much easier to understand than other available cpu options related to cpu-shares and cpu-sets.



`--cpus=<value>`	Specify how much of the available CPU resources a container can use. For instance, if the host machine has two CPUs and you set `--cpus="1.5"` , the container is guaranteed at most one and a half of the CPUs. This is the equivalent of setting `--cpu-period="100000"` and `--cpu-quota="150000"` . Available in Docker 1.13 and higher.

JVM MEMORY AND CPU MANAMGEMENT

JVM MEMORY FOOTPRINT

- So, what contributes to the JVM memory footprint? Most of us who have run a Java application, know how to set the maximum heap space. But there's actually a lot more contributing to the memory footprint:

- Native JRE
- Perm / metaspace
- JIT byte code
- JNI
- NIO
- Threads

- This is a lot that needs to be kept in mind when we want to set memory limits with Docker containers.

- And also setting the container memory limit to the maximum heap space, might not be sufficient.

JVM AND CPUS

- JVM adjust to the number of processors/cores available on the node it is running on.

- There are actually a number of parameters which by default are initialized based on core count:

- # of JIT compiler threads
- # Garbage Collection threads
- # of thread in the common fork-join pool
- ….

- So if the JVM is running on a 32 core node (and one did not overwrite the default), the JVM will spawn 32 Garbage Collection threads, 32 JIT compiler threads, ….

JVM MEETS CONTAINER

JVM MEETS CONTAINER

- Consider an example where everything works fine in development where developer is testing a single docker image on his laptop.

- In production, there are multiple docker containers for scalability. The operator/administrator allocate a CPUs to each docker container.

- Consider an example production node of 64 cores, with one CPU assigned to each docker container, there will be

- 10 * 64 Jit Compiler Threads
- 10 * 64 Garbage Collection threads
- 10 * 64 ….

- And our application, being limited in the number of cpu cycles it can use, is mostly dealing with switching between different threads and does cannot get any actual work done. Ouch..

JVM MEETS CONTAINER (CONTD…)

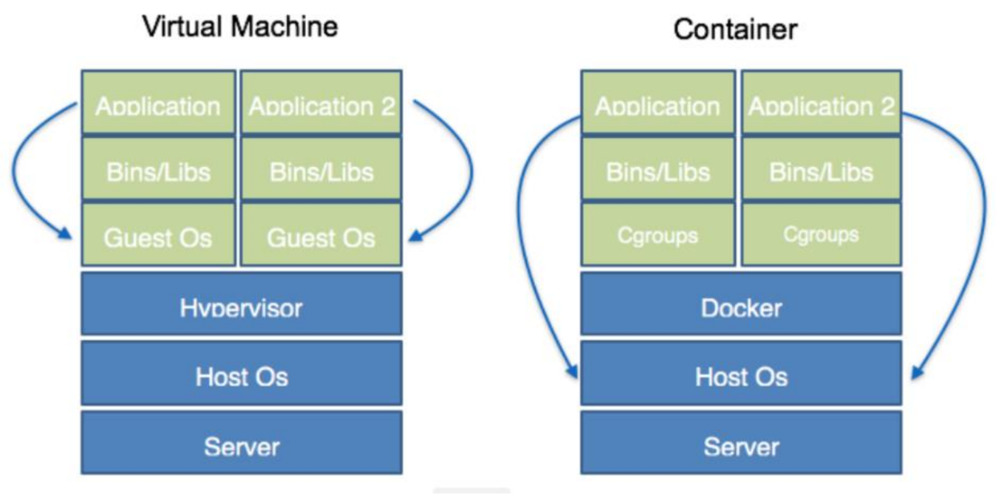
- Comparing VM with containers. On VMs, we will not see the same problem.

- In JDK 7/8, JVM gets the core count resources from sysconf. So JVM will see system's processors and not docker restriction.

- The same is true for default memory limits: the JVM will look at the host overall memory and use that for setting its defaults.

- So we can say that the JVM ignoring cgroups and that cause problems.

- Why namespaces are not coming to the rescue here? Unfortunately, there is no CPU or memory namespace.



SOLUTION

SOLUTION – MOVE TO JDK 10

- Move to JDK 10 which is container friendly.

- With JDK 10, Java will be self-aware about whether it is running inside a container or on a host.

- JDK 10 will introduce new JVM options for how containers control Memory and CPU settings. Both options will be used for configuring JVM inside containers so that it takes maximum heap size from Linux CGroup:

- **XX:-UseContainerSupport**—to allow container support to be disabled. Default: True i.e.: Enabled.

- **-XX:ActiveProcessorCount=xx**—this allows the number of CPUs to be overridden. This flag will be honored even if UseContainerSupport is not enabled.

SOLUTION – MOVE TO JDK 9; JVM SETTINGS

- Move to JDK 9 which supports JVM option for memory and handles CPU sets fine, but does not handle CPU shares.

- **-XX:+UseCGroupMemoryLimitForHeap**—to opt-in to using the value in /sys/fs/cgroup/memory/memory.limit_in_bytes as the value for phys_mem.

- **-XX:+UnlockExperimentalVMOptions**—enables experimental VM options supported for containers.

SOLUTION – MANUALLY SET JVM OPTIONS

- Manually overwrite the default parameter.

- E.g., at least XMX for memory and XX:ParallelGCThreads, XX:ConcGCThreads for CPU according to your specific cgroup limits.

CODE DEMO

CODE - DEMO

- Look at the HelloWorld.java which basically allocates memory in a tight loop

- There are two pre-built images – Java8 and Java10 which just executes the HelloWorld program on running. Images are based of JDK8 and JDK10.

- Run Java8 with memory (40 MB) and CPU (1 CPU) limit and see the # of cores seen by JVM as equal to the # of system's CPU and the OOM memory killer invocation in dmesg.

-Run Java10 with memory (40 MB) and CPU (1 CPU) limit and see the # of cores seen by JVM as 1 and the OOM memory killer not getting invoked in dmesg output.

-See https://github.com/jayrajput/meetup-bojug-java8-in-a-container for the demo code.

REFERENCES

1. Nobody Puts Java In A Container - Inspiration for this presentation
2. Java And Docker Memory Limits
3. Docker Intro
4. OpenJDK Bug which fixed the container issues
5. JAVA 10 Enhancements
6. JAVA SE support for docker CPU and Memory
7. Docker container configure resources
8. CPU management in docker 1.13
9. Blog inspiring the demo HelloWorld program
10. Demo Code in GitHub
11. Java Attacks Comic