Kubernetes on cgroup v2

Giuseppe Scrivano

During the talk

- Brief introduction of Linux cgroups
- How Kubernetes uses cgroups
- Introduce cgroup v2
- How we got to have cgroup v2 support in Kubernetes

What are cgroups?

- Cgroups are a kernel feature to restrict and monitor system resources usage of a group of processes
- Different specialized subsystems for each kind of resource
- They are hierarchical, each cgroup is restricted by the ancestor nodes
- Accessible from user space through a file system interface

How do we use them?

Cgroups are used by containers to solve problems like:

- Limit the container to use only two cpu cores
- Deny the container access to a particular device
- Give a container double CPU time than another
- Know how much memory the container is using

History

- Cgroup v1 developed at Google and merged into Linux 2.6.24 (24th Jan 2008)
- Cgroup v2 officially released with Linux 4.5 (13th Mar 2016)
- **Kubernetes Enhancement Proposal (PR #1370)** accepted in February 2020
- **Kubernetes 1.19** is the first release with cgroup v2 support

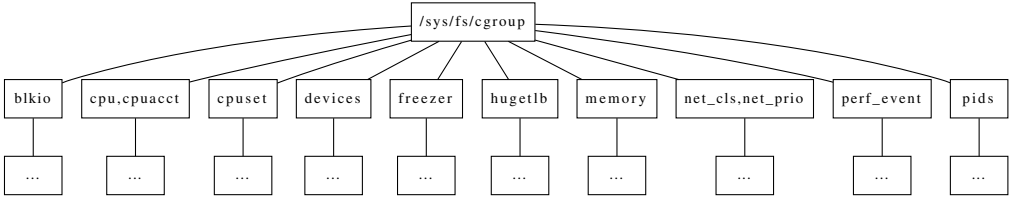
kernel version	cgroup v1 controllers added
2.6.24 (Jan 2008)	cpu, cpuacct, cpuset
2.6.25	memory
2.6.26	devices
2.6.28	freezer
2.6.29	netcls
2.6.33	blkio
2.6.39	perf_event
3.3	net_prio
3.5	hugetlb
4.3	pids
4.11 (April 2017)	rdma

kernel version	cgroup v2 controllers added
4.5 (March 2016)	io, memory, pids
4.11	perf_event, rdma
4.15	cpu
5.0	cpuset
5.2	freezer
5.6 (March 2020)	hugetlb

Cgroups v1

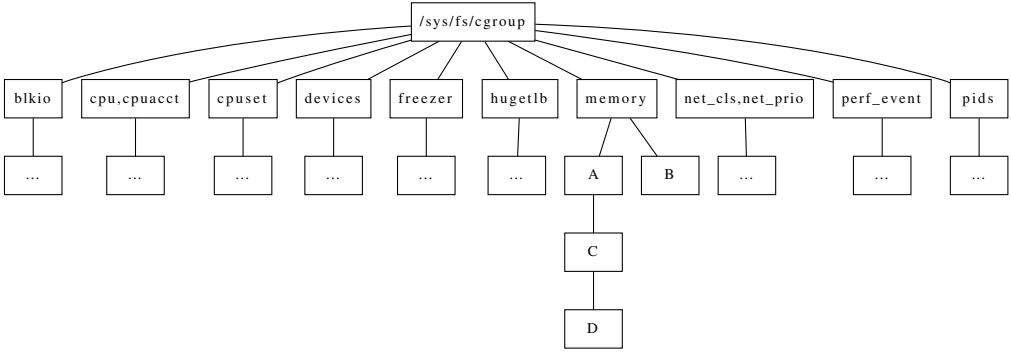
Cgroup v1

- Group of different controllers
- Each controller handles a single kind of resource
- Each controller is configured separately from the others



Cgroup v1

- Cgroup have a hierarchical structure
- Resources can be splitted further in the children cgroups
- A thread can belong to a different cgroup in each hierarchy



Cgroup API

- There are no syscalls or ioctls
- Fully controlled via the file system

Cgroup v1 mounts

- The cgroupfs virtual file system exposes all the API to the cgroups
- Each controller must be mounted separately
- Usually it is already mounted under */sys/fs/cgroup*

```
$ grep cgroup /proc/mounts
tmpfs /sys/fs/cgroup tmpfs ro,seclabel,nosuid,nodev,noexec,mode=755 0 0
cgroup /sys/fs/cgroup/systemd cgroup rw,seclabel,nosuid,nodev,noexec,relatime,xattr,name=systemd 0 0
cgroup /sys/fs/cgroup/cpu,cpuacct cgroup rw,seclabel,nosuid,nodev,noexec,relatime,cpu,cpuacct 0 0
cgroup /sys/fs/cgroup/memory cgroup rw,seclabel,nosuid,nodev,noexec,relatime,memory 0 0
cgroup /sys/fs/cgroup/cpuset cgroup rw,seclabel,nosuid,nodev,noexec,relatime,cpuset 0 0
cgroup /sys/fs/cgroup/freezer cgroup rw,seclabel,nosuid,nodev,noexec,relatime,freezer 0 0
cgroup /sys/fs/cgroup/net_cls,net_prio cgroup rw,seclabel,nosuid,nodev,noexec,relatime,net_cls,net_prio 0 0
cgroup /sys/fs/cgroup/perf_event cgroup rw,seclabel,nosuid,nodev,noexec,relatime,perf_event 0 0
cgroup /sys/fs/cgroup/blkio cgroup rw,seclabel,nosuid,nodev,noexec,relatime,blkio 0 0
cgroup /sys/fs/cgroup/pids cgroup rw,seclabel,nosuid,nodev,noexec,relatime,pids 0 0
cgroup /sys/fs/cgroup/hugetlb cgroup rw,seclabel,nosuid,nodev,noexec,relatime,hugetlb 0 0
cgroup2 /sys/fs/cgroup/unified cgroup2 rw,seclabel,nosuid,nodev,noexec,relatime,nsdelegate 0 0
```

Cgroup /proc files

Some information is exposed through the *procfs* file system

```
$ cat /proc/cgroups | column -t
#subsys_name  hierarchy  num_cgroups  enabled
cpuset        4          4            1
cpu           2          6            1
cpuacct       2          6            1
blkio         8          35           1
memory        3          114          1
devices       11         61           1
freezer       5          4            1
net_cls       6          4            1
perf_event    7          4            1
net_prio      6          4            1
hugetlb       10         4            1
pids          9          69           1
```

```
$ cat /proc/$PID/cgroup
11:hugetlb:/
10:freezer:/
9:memory:/user.slice/user-0.slice/session-1.scope
8:cpuset:/
7:devices:/user.slice
6:pids:/user.slice/user-0.slice/session-1.scope
5:blkio:/
4:net_cls,net_prio:/
3:perf_event:/
2:cpu,cpuacct:/
1:name=systemd:/user.slice/user-0.slice/session-1.scope
0::/user.slice/user-0.slice/session-1.scope
```

Cgroup v1 file system API

Mount a cgroup controller	# mount -t cgroup -omemory memory /sys/fs/cgroup/memory
Create a new cgroup	# mkdir /sys/fs/cgroup/memory/new-cgroup
Set a limit	# echo 1073741824 > /sys/fs/cgroup/memory/new-cgroup/memory.limit_in_bytes
Move a process PID to a cgroup	# echo $PID > /sys/fs/cgroup/memory/new-cgroup/cgroup.procs
Read some stats	# cat /sys/fs/cgroup/memory/new-cgroup/memory.max_usage_in_bytes
Read the processes in a cgroup	# cat /sys/fs/cgroup/memory/new-cgroup/cgroup.procs
Rename a cgroup	# mv /sys/fs/cgroup/memory/new-cgroup /sys/fs/cgroup/memory/different-name
Remove a cgroup (must be empty)	# rmdir /sys/fs/cgroup/memory/different-name

Quality of Service classes

- A QoS class is assigned to a pod

- There are 3 QoS classes
  - Guaranteed *CPU/Memory limits = CPU/Memory request*
    - Get what they ask for
  - Burstable *Not guaranteed but at least one CPU or Memory request*
    - Normal priority and what they requested (inside of the burstable cgroup)
    - Same OOM as guaranteed
  - Best Effort *Everything else*
    - Get the lowest priority
    - First to get killed when running low on memory

OOM killer

- Not strictly a cgroup feature
- It is used by Kubernetes to complement cgroups
- A task with a higher OOM score is killed first when there is not enough memory available

OOM score in Kubernetes

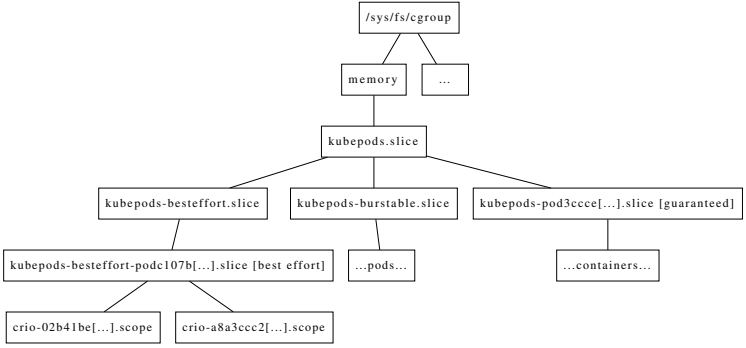
- Different OOM score for each QoS class

```
const (
        // KubeletOOMScoreAdj is the OOM score adjustment for Kubelet
        KubeletOOMScoreAdj int = -999
        // KubeProxyOOMScoreAdj is the OOM score adjustment for kube-proxy
        KubeProxyOOMScoreAdj  int = -999
        guaranteedOOMScoreAdj int = -998
        besteffortOOMScoreAdj int = 1000
)
```

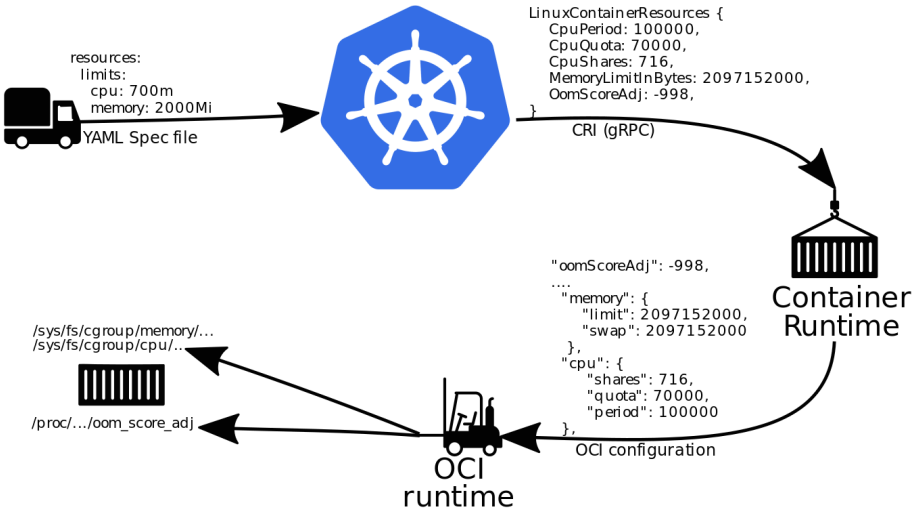
- Burstable pods gets a dynamic OOM score adjustment, that is between *guaranteedOOMScoreAdj* and *besteffortOOMScoreAdj*
- When there is not enough memory available, best effort pods are likely the first to be killed
- The Kubelet and kube-proxy run almost with the highest OOM score, but still killable
- Guaranteed pods are just one level below Kubelet and kube-proxy

How the Kubelet uses cgroup v1

- Cgroup are used together with the OOM score
- Limit resources isolations
- Monitoring



Running containers



- The Kubelet uses cAdvisor to monitor containers
- cAdvisor reads periodically stats from each cgroup
- It must understand the cgroup version used on the system
- On going work to move stats reading into the runtime

Some issues with cgroup v1

- Each controller must be handled separately. The flexibility of having different hierarchies is not used in practice
- Integration with some kernel subsystems, like *memory*, is not ideal
- Delegation of a subtree to a less privileged process is not safe
- No resources allocation
- Inconsistencies among the different subsystems
- Non atomic operations: creating, deleting, moving must be done for each controller
- The OOM is not cgroup aware, processes from different containers/cgroups can be killed at the same time

Cgroups v2

How we got here?

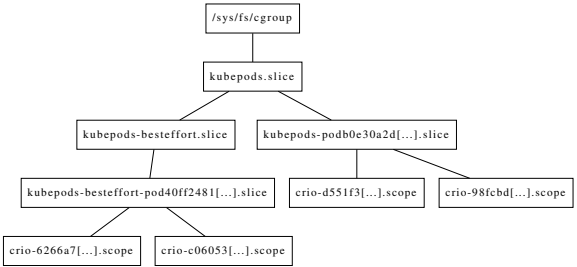
- Fedora 31, released October 2019, was the first distro to enable cgroup v2 by default
- crun was the first OCI runtime to support cgroup v2
- Podman uses crun by default on Fedora
- For running Kubernetes, it is necessary to switch to cgroup v1
- Hugetlb controller still missing in Linux when I've started working on cgroup v2

Why do we need cgroup v2?

- Cgroup v1 is considered legacy, no new features will be added
- Resources allocation is possible
- Cgroup aware OOM killer
- Delegation to less privileged processes is possible and safe
- Cgroup namespace enabled for unprivileged containers

Cgroup v2

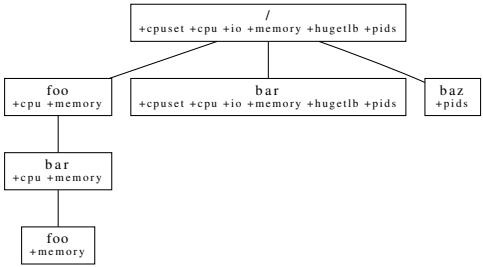
All the controllers are under the same hierarchy



- Delegation to less privileged processes is safe
- Some controllers, e.g. *devices*, require eBPF that is a privileged operation

Cgroup v2 controllers

- The controllers are a property of the cgroup



- A cgroup can use a controller only if it is enabled in the parent cgroup
- A controller is enabled or disabled through the *cgroup.subtree_control* file

```
# echo +cpu > /sys/fs/cgroup/parent/cgroup/cgroup.subtree_control
# echo -cpu > /sys/fs/cgroup/parent/cgroup/cgroup.subtree_control
```

Cgroup v2 controllers

cgroup v1	cgroup v2	can be unprivileged
blkio	io	✓
cpu,cpustat	cpu	✓
cpuset	cpuset	✓
devices	with eBPF	✗
freezer	freezer	✓
net_cls,net_prio	with eBPF	✗
hugetlb	hugetlb	✓
perf_event	perf_event	✓
pids	pids	✓
rdma	rdma	✓

Some other differences:

- Not everything that is available in cgroup v1 is present in cgroup v2 (e.g. *cpuacct.usage_percpu*).
- Cgroup v2 only features (e.g. Pressure Stall Information).
- Different semantic (e.g. swap memory limits).
- Different ranges (e.g. cpu.shares uses [2-262144], cpu.weight uses [1-10000]).

Cgroup v2 rules

- Processes/threads can be added only to leaf nodes.
- Before a cgroup can use a controller, the controller must be enabled for all the parent cgroups
- Originally all the threads in a process had to be in the same cgroup. Relaxed in newer Linux versions.
- The *nsdelegate* mount option makes delegation safe. More restrictions apply in the cgroup namespace.

Pressure Stall Information metrics

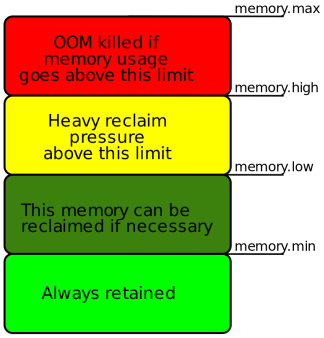
- Cgroup v2 gives new metrics to detect resource shortages
- Metrics for IO, CPU and memory resources
- Percentage of wall time spent waiting for a resource for *some* or *all* processes
- Record it for the last 10 seconds, 1 minute and 5 minutes
- Also show the accumulated time in microseconds

```
$ cat /sys/fs/cgroup/system.slice/io.pressure
some avg10=0.00 avg60=0.00 avg300=0.00 total=5461793
full avg10=0.00 avg60=0.00 avg300=0.00 total=3646744
```

Memory Protection

Memory is configured with 4 files

- *memory.min* It will never be reclaimed
- *memory.low* Soft protection, memory below this threshold is reclaimed only if there is nothing reclaimable in other cgroups
- *memory.high* Memory usage throttling, the kernel tries to keep the memory usage below this limit
- *memory.max* Hard limit, the OOM killer is invoked on the cgroup if trying to use more memory than this limit



Cgroup namespace

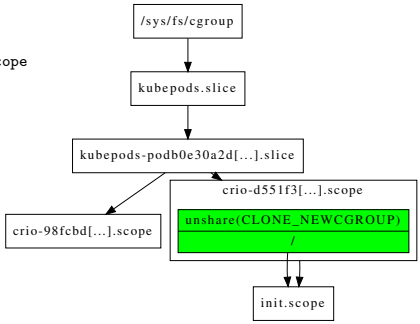
Without a cgroup namespace

```
# cat /proc/self/cgroup
0::/system.slice/run-rbeb7749cb8ad4be486f4ba0b59a81e50.scope
```

Within a cgroup namespace

```
# unshare -C cat /proc/self/cgroup
0::/
```

- A cgroup namespace turns the current cgroup into the cgroup root
- While in a cgroup namespace, it is not possible to move processes out of it



Open Container Initiative containers

OCI runtime specifications

- The OCI runtime is responsible for the final container setup
- Lowest level in the stack, just above the kernel
- A JSON file *config.json* is used to describe the container

OCI runtime specifications

The *resources* object specifies how to configure cgroups.

```
{
    "resources": {
        "memory": {
            "limit": 2097152000,
            "swap": 2097152000
        },
        "cpu": {
            "shares": 716,
            "quota": 70000,
            "period": 100000
        },
        "pids": {
            "limit": 1024
        },
        "hugepageLimits": [
            {
                "pageSize": "1GB",
                "limit": 0
            },
            {
                "pageSize": "2MB",
                "limit": 0
            }
        ]
    }
}
```

Each resource is mapped to the cgroup file system

2097152000 → /sys/fs/cgroup/**memory**/$CGROUP/memory.limit_in_bytes
2097152000 → /sys/fs/cgroup/**memory**/$CGROUP/memory.memsw.limit_in_bytes

716 ⟶ /sys/fs/cgroup/**cpu**/$CGROUP/cpu.shares
70000 ⟶ /sys/fs/cgroup/**cpu**/$CGROUP/cpu.cfs_quota_us
100000 ⟶ /sys/fs/cgroup/**cpu**/$CGROUP/cpu.cpu.cfs_period_us

1024 → /sys/fs/cgroup/**pids**/$CGROUP/pids.max

0 ⟶ /sys/fs/cgroup/**hugetlb**/$CGROUP/hugetlb.1GB.limit_in_bytes

0 ⟶ /sys/fs/cgroup/**hugetlb**/$CGROUP/hugetlb.2MB.limit_in_bytes

OCI runtime specifications

The OCI runtime specifications are designed for cgroup v1

Some issues with cgroup v2:

- Different file names
- Different ranges for the values
- The specs are not extendable
  - Cannot support new features

OCI runtime specifications

The OCI runtime specifications are designed for cgroup v1

Some issues with cgroup v2:

- Different file names ✓
- Different ranges for the values ✓
- The specs are not extendable ✗ Proposal for cgroup v2 support
  - Cannot support new features

Conversion from cgroup v1 to cgroup v2

- crun implemented by to get started
- Attempt a conversion from the cgroup v1 configuration to cgroup v2

Example

The *cpu.shares* value is converted from *[2-262144]* to the range *[1-10000]* accepted for *cpu.weight*.

- A container engine, unless it is using directly the cgroup, won't need to generate a different OCI configuration
- Now runc performs the same conversions

What is next?

What is next?

- Extend OCI to support new cgroup v2 features
- Take advantage of the cgroup v2 memory allocation
- Use the new *Pressure Stall Information metrics*
- *Enable running Kubernetes without root privileges*
- *Nested Kubernetes?*


KubeCon
CloudNativeCon
Europe 2020
Virtual
HELM
KEEP CLOUD NATIVE
CONNECTED