How to Evolve Kubernetes Resource Management Model

Jiaying Zhang (github.com/jiayingz)
June 26th, 2019

kubernetes

Why you may want to listen to this talk as an app developer

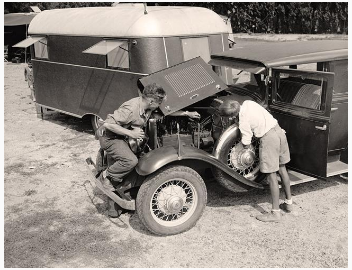
You know how to use
it when you see it

⚠ WARNING

Read and fully understand operator's manual before using this machine.

Failure to follow operating instructions could result in death or serious injury.

Need to read user
manual, *carefully*



Need to understand
some underlying
mechanisms to operate

where we are today

Evolving
Kubernetes
Resource
Management
Model

Why do I need Kubernetes and what can it do - from Kubernetes Concepts

- Service discovery and load balancing
- Storage orchestration
- Automated rollouts and rollbacks
- **Automatic bin packing**

  Kubernetes allows you to specify how much CPU and memory (RAM) each container needs. When containers have resource requests specified, Kubernetes can make better decisions to manage the resources for containers.
- Self-healing
- Secret and configuration management

Why do I need to care about resource management in Kubernetes?

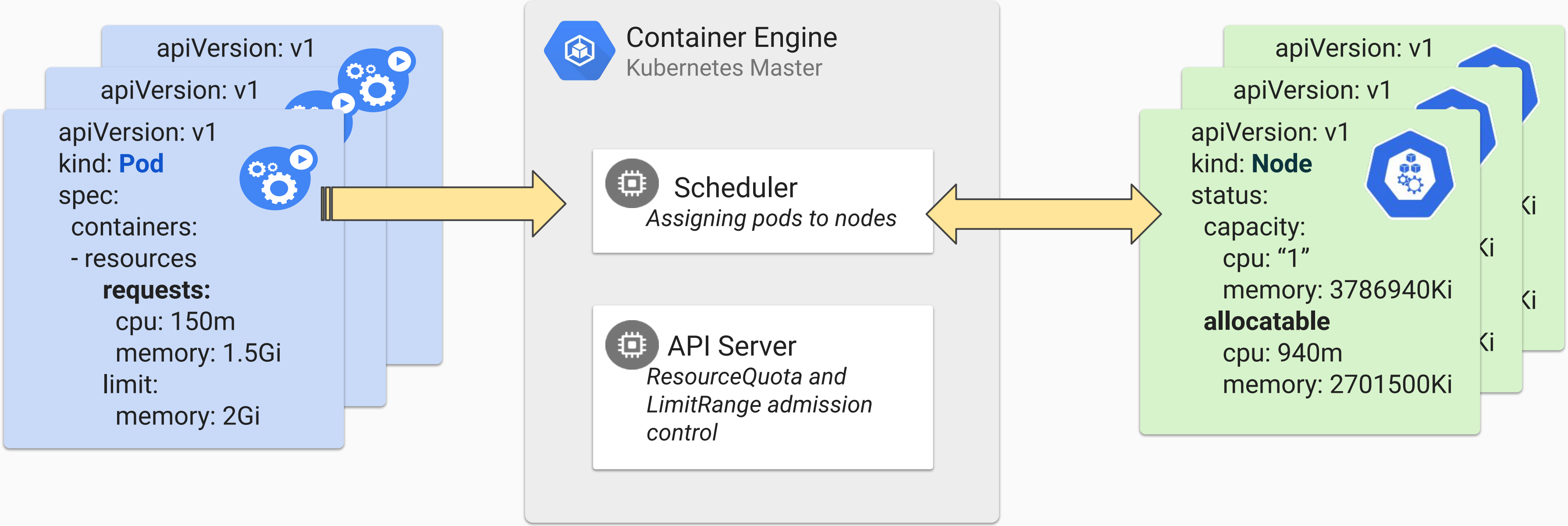
- Resource efficiency is one of major benefits of Kubernetes
- People want their applications to have predictable performance
- Some underlying details you want to know to make better use of your resources and avoid future pitfalls

Let’s start with a simple web app

```
metadata:
  name: myapp
spec:
  containers:
  - name: web
  - resources
      requests:
        cpu: 300m
        memory: 1.5Gi
      Limits:
        cpu: 500m
        memory: 2Gi
```

```
$ kubectl create -f myapp.yaml
pod "myapp" created

$ kubectl get pod myapp
NAME      READY     STATUS     RESTARTS    AGE
myapp     0/1       Pending    0           29s

$ kubectl describe pod myapp
Name:           myapp
Namespace:      default
Node:           <none>
…
Events:
  Type            Reason                  Message
  Warning         FailedScheduling        0/3 nodes
are available: 3 Insufficient memory.
```

High level overview



apiVersion: v1

apiVersion: v1

```
apiVersion: v1
kind: Pod
spec:
  containers:
  - resources
      requests:
        cpu: 150m
        memory: 1.5Gi
      limit:
        memory: 2Gi
```

Container Engine
Kubernetes Master

Scheduler
Assigning pods to nodes

API Server
ResourceQuota and LimitRange admission control

apiVersion: v1

apiVersion: v1

```
apiVersion: v1
kind: Node
status:
  capacity:
    cpu: “1”
    memory: 3786940Ki
  allocatable
    cpu: 940m
    memory: 2701500Ki
```

Scheduler - assign node to pod

- A very simplified view from 1000 feet high:

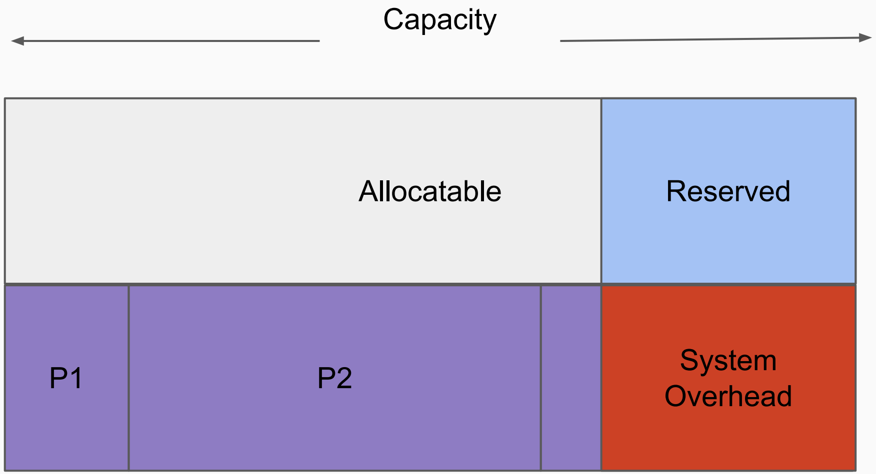
```
while True:
    pods = get_all_pods()
    for pod in pods:
        if pod.node == nil:
           assignNode(pod)
```

- Scheduling algorithm makes sure selected node satisfies pod resource requests
    - For each specified resource, ∑Pod **requests** <= node **allocatable**

Node level

System processes also compete resources with user pods

- Allocatable resource
  - how much resources can be allocated to users' pods
  - allocatable = capacity - reserved (system overhead)



Reserve enough resources for system components to avoid problems when utilization is high

Pod requested resource needs to be within node allocatable

```
metadata:
  name: myapp
spec:
  containers:
  - name: web
  - resources
      requests:
        cpu: 300m
        memory: 1.5Gi
      Limits:
        cpu: 500m
        memory: 2Gi
```

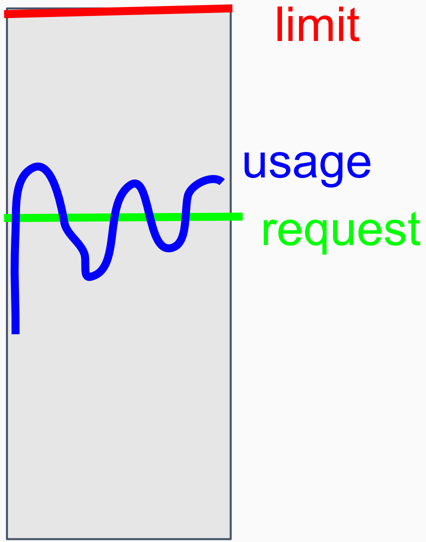
```
# create a node with more memory

$ kubectl get pod myapp
NAME        READY       STATUS      RESTARTS     AGE
myapp       1/1         Running     0            4s

$ kubectl describe pod myapp
Name:               myapp
Namespace:          default
Node:               node1
…
Events:
  Type                Reason                         Message
Scheduled           Successfully assigned default/myapp to node1
...
Created             Created container
Started             Started container
```

What about limits? - Limits are only used at node level

- Desired State (specification)
  - request: amount of resources requested by a container/pod
  - limit: an upper cap on the resources used by a container/pod
- Actual State (status)
  - actual resource usage: lower than limit

Based on request/limit setting, pods have different QoS

- Guaranteed: 0 < request == limit
- Burstable: 0 < request < limit
- Best effort: no request/limit specified, lowest priority



But you need to know a bit more to use them right

Resource requests and limits can have different implications on different resources, as the underlying enforcing mechanisms are different.

- Compressible
    - Can be throttled
    - “Merely” cause slowness when revoked
    - E.g., CPU, network bandwidth, disk IO
- Incompressible
    - Not easily throttled
    - When revoked, container may die or pod may be evicted
    - E.g., memory, disk space, no. of processes, inodes

How CPU requests are used at node

- CPU requests map to cgroup cpu.shares
- CPU share defines relative CPU time assigned to a cgroup
  - cgroup assigned cpu time = cpu.shares / total_shares
  - E.g., 2 available cpu cores, c1: 200 shares, c2: 400 shares
    - c1: 0.67 cpu time, c2: 1.33 cpu time
  - E.g., 2 available cpu cores, c1: 200 shares, c2: 400 shares, c3: 200 shares
    - c1: 0.5 cpu time, c2: 1 cpu time, c3: 0.5 cpu time

```yaml
    resources:
      requests:
        cpu: 300m
      limits:
        cpu: 500m
```

```
$ cat /sys/fs/cgroup/cpu/kubepods/burstable/podxxx/cpu.shares
307
```

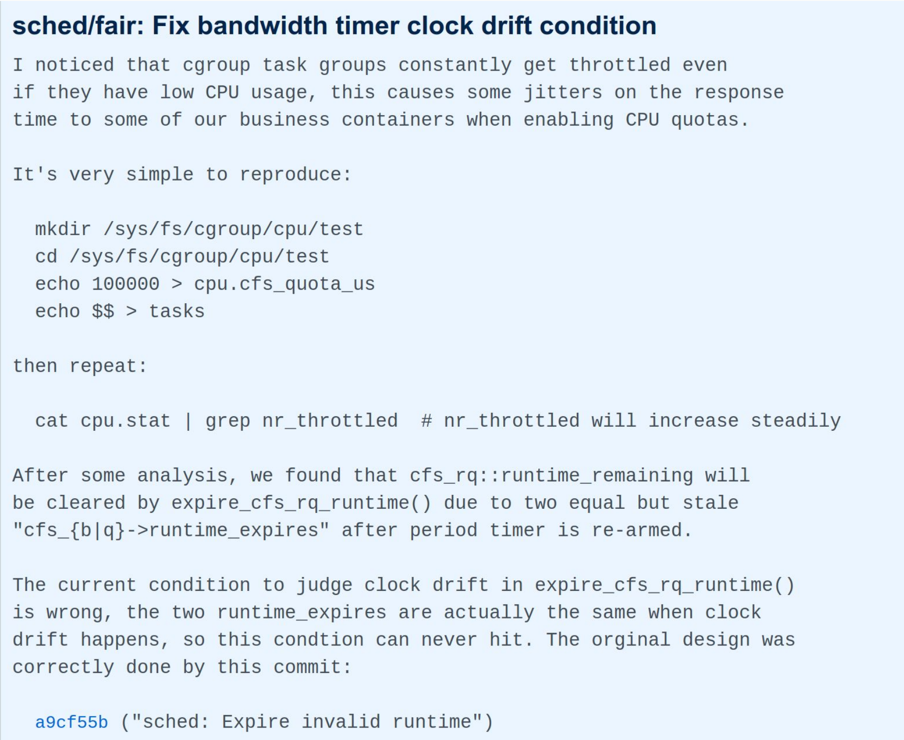
How CPU limits are used at node

- CPU limits map to cgroup cfs “quota” in each given “period”
  - cpu.cfs_quota_us: the total available run-time within a period
  - cpu.cfs_period_us: the length of a period. Default setting: 100ms.
- Implication: can cause latency if not set correctly
- E.g.: a container takes 30ms to handle a request without throttling
  - 50m cpu limit: takes 30ms to finish the task
  - 20m cpu limit: takes > 100ms to finish the task

```yaml
resources:
  requests:
    cpu: 300m
  limits:
    cpu: 500m
```

```
$ cat /sys/fs/cgroup/cpu/kubepods/burstable/podxxx/cpu.cfs_quota_us
50000
$ cat /sys/fs/cgroup/cpu/kubepods/burstable/podxxx/cpu.cfs_period_us
100000
```

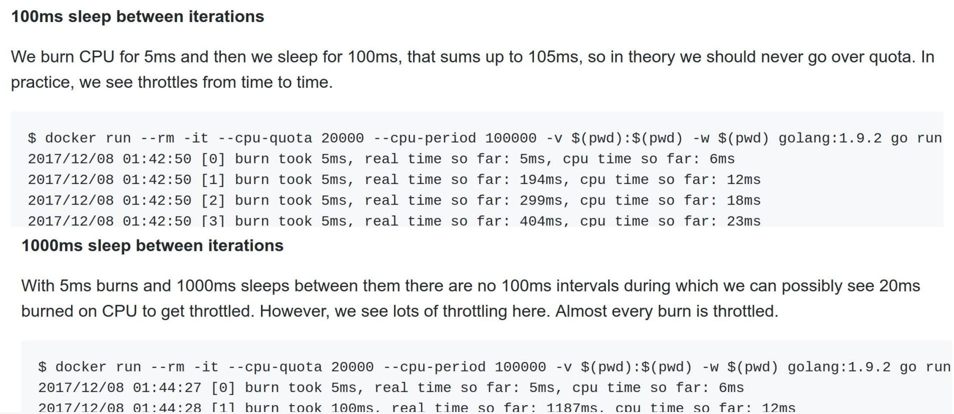
Caveats on using cpu limits - example issues on completely fair scheduler (CFS)



sched/fair: Fix bandwidth timer clock drift condition

```
I noticed that cgroup task groups constantly get throttled even
if they have low CPU usage, this causes some jitters on the response
time to some of our business containers when enabling CPU quotas.

It's very simple to reproduce:

  mkdir /sys/fs/cgroup/cpu/test
  cd /sys/fs/cgroup/cpu/test
  echo 100000 > cpu.cfs_quota_us
  echo $$ > tasks

then repeat:

  cat cpu.stat | grep nr_throttled  # nr_throttled will increase steadily

After some analysis, we found that cfs_rq::runtime_remaining will
be cleared by expire_cfs_rq_runtime() due to two equal but stale
"cfs_{b|q}->runtime_expires" after period timer is re-armed.

The current condition to judge clock drift in expire_cfs_rq_runtime()
is wrong, the two runtime_expires are actually the same when clock
drift happens, so this condtion can never hit. The orginal design was
correctly done by this commit:

  a9cf55b ("sched: Expire invalid runtime")
```

Overly aggressive CFS



100ms sleep between iterations

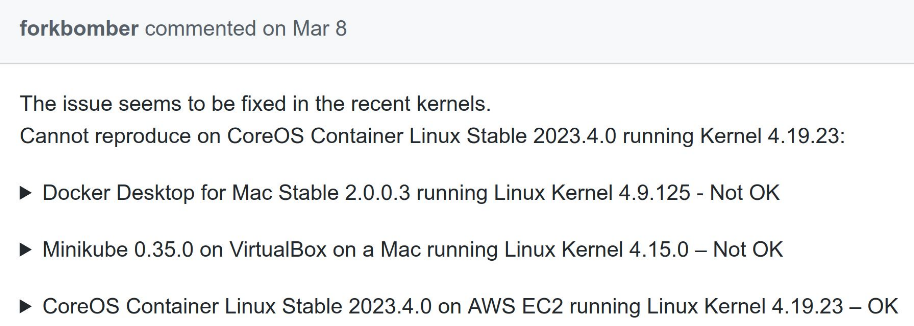
We burn CPU for 5ms and then we sleep for 100ms, that sums up to 105ms, so in theory we should never go over quota. In practice, we see throttles from time to time.

```
$ docker run --rm -it --cpu-quota 20000 --cpu-period 100000 -v $(pwd):$(pwd) -w $(pwd) golang:1.9.2 go run
2017/12/08 01:42:50 [0] burn took 5ms, real time so far: 5ms, cpu time so far: 6ms
2017/12/08 01:42:50 [1] burn took 5ms, real time so far: 194ms, cpu time so far: 12ms
2017/12/08 01:42:50 [2] burn took 5ms, real time so far: 299ms, cpu time so far: 18ms
2017/12/08 01:42:50 [3] burn took 5ms, real time so far: 404ms, cpu time so far: 23ms
```

1000ms sleep between iterations

With 5ms burns and 1000ms sleeps between them there are no 100ms intervals during which we can possibly see 20ms burned on CPU to get throttled. However, we see lots of throttling here. Almost every burn is throttled.

```
$ docker run --rm -it --cpu-quota 20000 --cpu-period 100000 -v $(pwd):$(pwd) -w $(pwd) golang:1.9.2 go run
2017/12/08 01:44:27 [0] burn took 5ms, real time so far: 5ms, cpu time so far: 6ms
2017/12/08 01:44:28 [1] burn took 100ms, real time so far: 1187ms, cpu time so far: 12ms
```



forkbomber commented on Mar 8

The issue seems to be fixed in the recent kernels.
Cannot reproduce on CoreOS Container Linux Stable 2023.4.0 running Kernel 4.19.23:

▶ Docker Desktop for Mac Stable 2.0.0.3 running Linux Kernel 4.9.125 - Not OK

▶ Minikube 0.35.0 on VirtualBox on a Mac running Linux Kernel 4.15.0 – Not OK

▶ CoreOS Container Linux Stable 2023.4.0 on AWS EC2 running Linux Kernel 4.19.23 – OK

Understand why you want to use cpu limits

- Pay-per-use: constraint cpu usage to limit cost
- Latency provisioning: set latency expectations with worst-case CPU access time
- Reserve exclusive cores: static CPU manager
- Keep Pod in guaranteed QoS to avoid:
    - Eviction: no longer based on QoS class any more
    - OOM killing: still takes QoS into account, but you perhaps want to avoid OOM killing by setting your memory requests/limits right

Quick takeaway: if you have to use CPU limits, use it with care

How memory requests are used at node

- Memory requests don’t map to cgroup setting.
- They are used by Kubelet for memory eviction.

```
$ kubectl describe pod myapp
Name:               myapp
…
Events:
  Type               Reason                   Message
Scheduled          Successfully assigned default/myapp to node1
...
Created            Created container
Started            Started container
     Evicted            The node was low on resource: memory. Container myapp was using 12700Ki,
which exceeds its request of 5000Ki
     Killing            Killing container with id docker://myapp:Need to kill Pod
```

```
metadata:
  name: myapp
spec:
  containers:
  - resources
      requests:
        memory: 5Mi
      Limits:
        memory: 20Mi
```

Eviction - Kubelet’s hammer to reclaim incompressible resources

- Kubelet determines when to reclaim resources based on eviction signals and eviction thresholds
- Eviction signal: current available capacity of a resource. What we have today:
  - `memory.available & allocatableMemory.available`
  - `nodefs.available & imagefs.available`
  - `nodefs.inodesFree & imagefs.inodesFree`
  - `pid.available -` *`partially implemented`*
- Eviction threshold: minimum value of **a resource** Kubelet should maintain
  - Eviction-soft is hit: Kubelet starts reclaiming resource with Pod termination grace period as **`min(eviction-max-pod-grace-period, pod.Spec.TerminationGracePeriod)`**
  - Eviction-hard is hit: Kubelet starts reclaiming resources immediately, without grace period.

Eviction - Kubelet’s hammer to reclaim incompressible resources

- Kubelet determines when to reclaim resources based on eviction signals and eviction thresholds
- Eviction signal: current available capacity of a resource. What we have today:
  - `memory.available & allocatableMemory.available`
  - `nodefs.available & imagefs.available`
  - `nodefs.inodesFree & imagefs.inodesFree`
  - `pid.available (partially implemented)`
- ***Ideally, your providers/operators should set these configs right for you that you need to worry about them.***
- Eviction threshold: minimum value of **a resource** Kubelet should maintain
  - Eviction-soft is hit: Kubelet starts reclaiming resource with Pod termination grace period as **`min(eviction-max-pod-grace-period, pod.Spec.TerminationGracePeriod)`**
  - Eviction-hard is hit: Kubelet starts reclaiming resources immediately, without grace period.

What you need to know about eviction?

- Your pod may get evicted when it uses more than its requested amount of a resource and that resource is near being exhausted on a node
- Kubelet decides which pod to evict based on eviction score calculated from:
  - Pod priority
  - How much pod’s actual usage is above its requests

    Caveat: currently not implemented for pid.

What you need to know about eviction?

- You can reduce your pod’s risk of being evicted by:
  - Set right requests for memory and ephemeral storage.
  - Avoid using too much of other types of incompressible resources or increase their node limits.
  - Using higher priority.

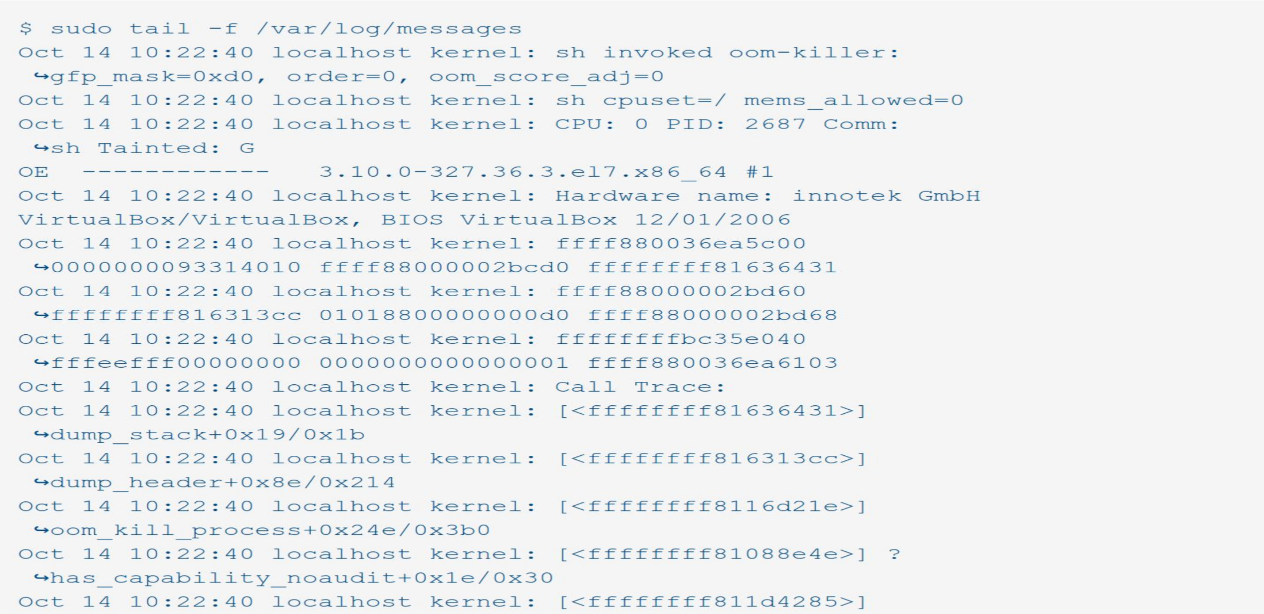
What you need to know about eviction?

- When things go unexpected, check with cluster operator on the underlying settings
    - Kubelet or Docker run out of a resource: resource eviction signal and threshold settings
    - Frequently exhausts pids or inodes: Node sysctl setting
    - Pod terminates too quickly: eviction max pod grace period setting
    - Node oscillating on resource pressure (e.g., MemoryPressure, DiskPressure) conditions: eviction pressure transition period setting

How memory limits are used at node

- Memory limits map to cgroup memory.limit_in_bytes
- Container exceeding its memory limits will get OOM-killed

```
$ cat /sys/fs/cgroup/memory/kubepods/burstable/podxxx/memory.limit_in_bytes
134217728
```



```
$ sudo tail -f /var/log/messages
Oct 14 10:22:40 localhost kernel: sh invoked oom-killer:
 ↪gfp_mask=0xd0, order=0, oom_score_adj=0
Oct 14 10:22:40 localhost kernel: sh cpuset=/ mems_allowed=0
Oct 14 10:22:40 localhost kernel: CPU: 0 PID: 2687 Comm:
 ↪sh Tainted: G
OE  ------------   3.10.0-327.36.3.el7.x86_64 #1
Oct 14 10:22:40 localhost kernel: Hardware name: innotek GmbH
VirtualBox/VirtualBox, BIOS VirtualBox 12/01/2006
Oct 14 10:22:40 localhost kernel: ffff880036ea5c00
 ↪0000000093314010 ffff88000002bcd0 ffffffff81636431
Oct 14 10:22:40 localhost kernel: ffff88000002bd60
 ↪ffffffff816313cc 01018800000000d0 ffff88000002bd68
Oct 14 10:22:40 localhost kernel: ffffffffbc35e040
 ↪fffeefff00000000 0000000000000001 ffff880036ea6103
Oct 14 10:22:40 localhost kernel: Call Trace:
Oct 14 10:22:40 localhost kernel: [<ffffffff81636431>]
 ↪dump_stack+0x19/0x1b
Oct 14 10:22:40 localhost kernel: [<ffffffff816313cc>]
 ↪dump_header+0x8e/0x214
Oct 14 10:22:40 localhost kernel: [<ffffffff8116d21e>]
 ↪oom_kill_process+0x24e/0x3b0
Oct 14 10:22:40 localhost kernel: [<ffffffff81088e4e>] ?
 ↪has_capability_noaudit+0x1e/0x30
Oct 14 10:22:40 localhost kernel: [<ffffffff811d4285>]
```

```
resources:
  limits:
    memory: 128Mi
```

Why you may still see OOM killing without exceeding your limits

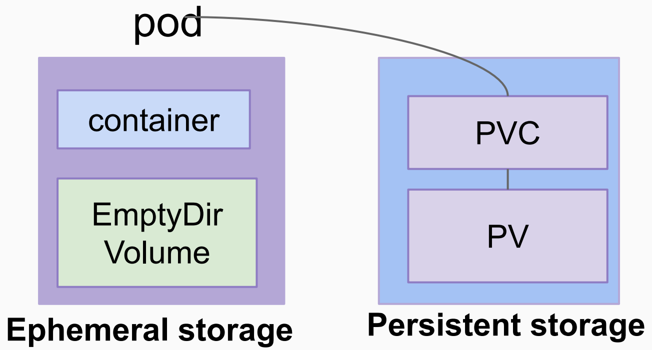
- OS can kick in before Kubelet is able to reclaim enough memory - OOM killing
- Under memory pressure, Linux kernel determines which process to kill based on *oom_score*
- Today, Kubelet adjusts oom_score based on QoS class and memory requests:
  - Critical node components (Kubelet, Docker, etc): `-999`
  - Guaranteed Pod: `-998`
  - Best-effort Pod: `1000`
  - Burstable Pod: between `-998 to 1000`, calculated based on memory requests

What you need to know about OOM killing?

- OOM killing is even worse than memory eviction
    - You whole system may experience performance downgrade
    - Application doesn’t have chance to terminate gracefully
- You can reduce chance for your application being OOM killed by:
    - Setting right memory limits
    - Reserve enough memory for your system components
    - Don’t accumulate too many dirty pages

Local ephemeral storage - Beta

- Local ephemeral: local root partition **shared** by pods/containers and system components
  - Same lifetime as pods/containers
  - Container: writable layers, image layers, logs
  - Pod: emtyDir volumes
- Persistent: dedicated disks (remote or local)
  - Explicit lifetime outlives containers/pods
  - Represented by PV/PVC



```
apiVersion: v1
kind: Pod
spec:
  containers:
  - name: db
    image: mysql
   volumeMounts:
      - mountPath: /cache
        name: cache-volume
   volumeMounts:
      - mountPath: /database
        name: database-volume
   volumes:
    - name: cache-volume
      emptyDir:{}
   volumes:
    - name: database-volume
      persistentVolumeClaim:
         claimName: task-pv-claim
```

How to set ephemeral storage resource requirements

- Container level: can specify ***ephemeral-storage*** requests and limits
- Pod level: emptyDir ***sizeLimit***
- Scheduler schedules a Pod to a node if the sum of the ephemeral-storage requests from the scheduled containers is less than the node’s allocatable ephemeral-storage

```
apiVersion: v1
kind: Pod
metadata:
  name: frontend
spec:
  containers:
  - name: db
    image: mysql
    resources:
      requests:
        ephemeral-storage: "2Gi"
      limits:
        ephemeral-storage: "4Gi"
 volumeMounts:
    - mountPath: /cache
      name: cache-volume
 volumes:
  - name: cache-volume
    emptyDir:
       sizeLimit: “10Gi”
```

Ephemeral storage eviction

- Under disk pressure, a pod can get evicted if:
  - With *LocalStorageCapacityIsolation* enabled:
    - It has a container whose ephemeral storage usage exceeds the container’s limits
    - It has an emptyDir whose disk usage exceeds its sizeLimit
    - ∑ container’s usage + ∑ emptyDir’ usage > ∑ container’s limits
  - It has highest eviction score calculated from:
    - Priority
    - How much pod’s actual usage is above its requests

Beyond basic use cases

- What if my app makes heavy use of disk IO?
  - Provision enough IO bandwidth and IOPs on your node
  - Avoid running two IO heavy Pods on the same node with Pod anti-affinity
  - Consider to use dedicated disks/volumes
- What if my app is network latency sensitive or requires a lot network bandwidth?
  - Use Pod anti-affinity to spread your pods to different nodes
  - Can request high-performance NIC as extended resource
  - *but* first make sure bottleneck is not on network switches

Beyond basic use cases

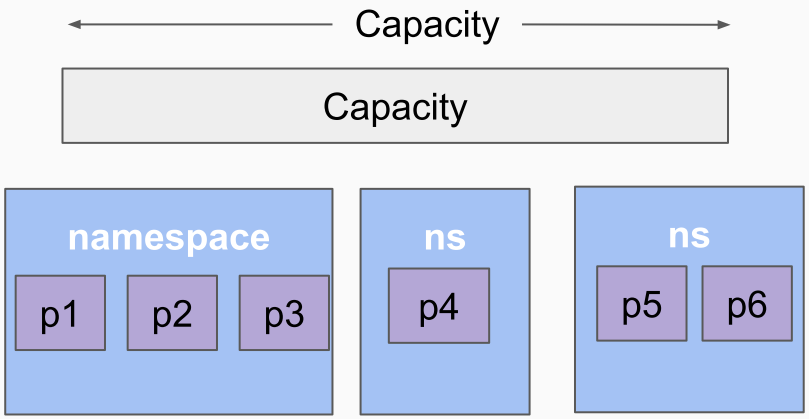
- What if my app is sensitive to CPU cache interference
    - Use static CPU manager policy and request integer number of CPUs
- What if I want to run my workload on GPU?
    - Can request GPU as **extended resource**, with requests == limits
    - Better protect your GPU resource with taints & tolerations

Other things may affect your pod’s scheduling/running

- Priority and preemption
  - Preempt lower priority pods to schedule higher priority pending pods
  - Knob to make sure your high-priority workload have place to run.
- Resource Quota admission
- LimitRange

Resource admission control - how different teams share resources in a cluster

- Namespace
  - Partition resources into logically named groups
  - Ability to specify resource constraints for each group



Resource admission control - how different teams share resources in a cluster

- Resource quota: specifies total resource requests/limits for a namespace
  - Checked during pod creation through API server admission control:
    - ∑Pod requests <= request quota
    - ∑Pod limit <= limit quota

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: demo
spec:
  hard:
    requests.cpu: 5
  scopeSelector:
    matchExpressions:
    - operator: In
      scopeName: PriorityClass
       Values: [“low”]
```

Resource admission control - how different teams share resources in a cluster

- LimitRange
    - Configures default requests and limits for a namespace
    - Enforce minimum/maximum pod/container resource requirements
    - Enforce a ratio between request and limit for a resource

```yaml
apiVersion: v1
kind: LimitRange
metadata:
  name: demo
spec:
  limits:
  - default:
      cpu: 500m
      Memory: 900Mi
    defaultRequest:
      cpu: 100m
      Memory: 100Mi
  type: Container
```

Too many things to think about?

Things that can make your life easier - Horizontal Pod Autoscaler (HPA)

- Automatically scale up/down pods in a ReplicaSet based on CPU utilization or some metrics you defined
- Use HPA when
  - You can load balance work among replicas
  - Your pod's resource usage is proportional to its work input
  - Better to be combined with Cluster Autoscaler

Things that can make your life easier - Cluster Autoscaler (CA)

- Add more nodes to run pending pods or scale down node after your job finishes
- Use CA if nodes can be dynamically created in your k8s cluster

Things that can make your life easier - Vertical Pod Autoscaler (VPA)

- Measures and/or sets resource requests for you.
- Consider VPA if your application's resource requirements change over time
- Bearing in mind some of its features are still experimental

Wrap up

- Set CPU requests to reserve CPU time your pod needs. Use CPU limits with care.
- Sets correct memory requests/limits to avoid memory eviction and/or OOM.
- Prevents your nodes from running out of disk with ephemeral storage requests/limits and emptyDir sizeLimit.
- Avoid exhausting incompressible resources.
- If your pod uses a lot IO or network, try to provision enough or not share them.
- Understand your cluster admin setting to avoid surprise.
- You can request GPU as extended resource.
- Use autoscalers if possible to make your life easier.

We still have a LONG way to go

Evolving
Kubernetes Resource
Management
Model