KUB 15: Планирование запуска pods

Описание:

Requests / Limits
Серьезным аспектом работы планировщика в Kubernetes являются ресурсы. Для более полной утилизации хостов планировщик обязан знать объем существующих ресурсов и запросы на эти ресурсы. Все это прописывается в описании контейнера.

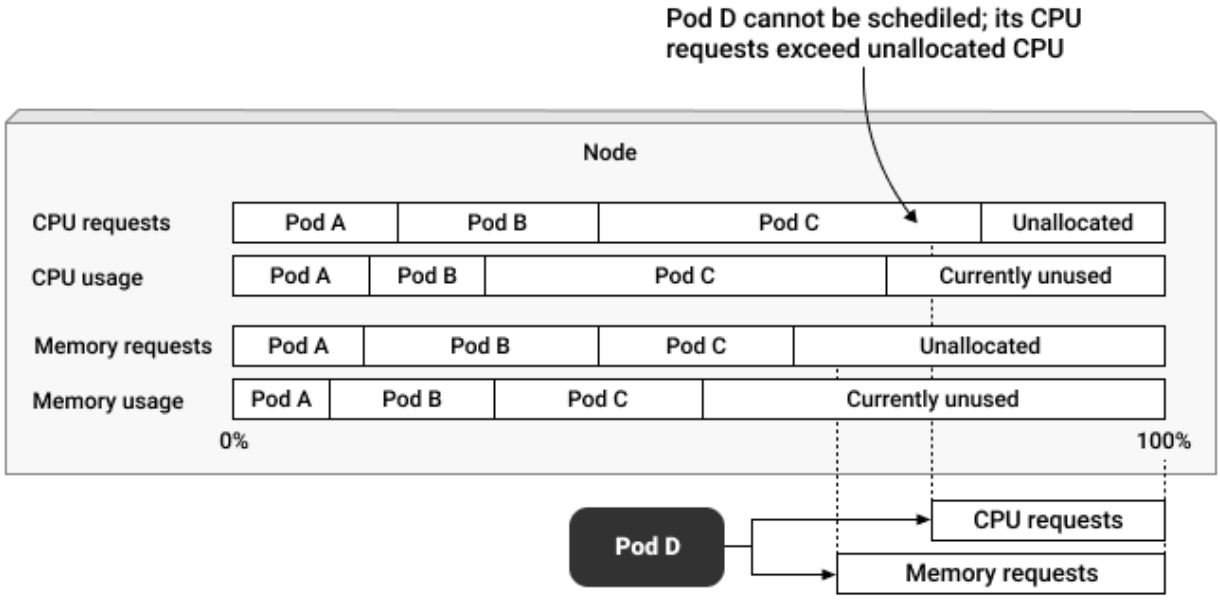
```
apiVersion: v1
kind: Pod
metadata:
  name: requests-pod
spec:
  containers:
  - image: busybox
    command: ["dd", "if=/dev/zero", "of=/dev/null"]
    name: main
    resources:
      requests:
        cpu: "200m"
        memory: "10Mi"
```

В примере выше мы запрашиваем 10 мегабайт памяти и 200 миллиядер процессора (1/5 ядра). Если применить такую конфигурацию и запустить после этого в контейнере утилиту top, то мы увидим, что использовано более 50% cpu, а это никак не соответствует 1/5.

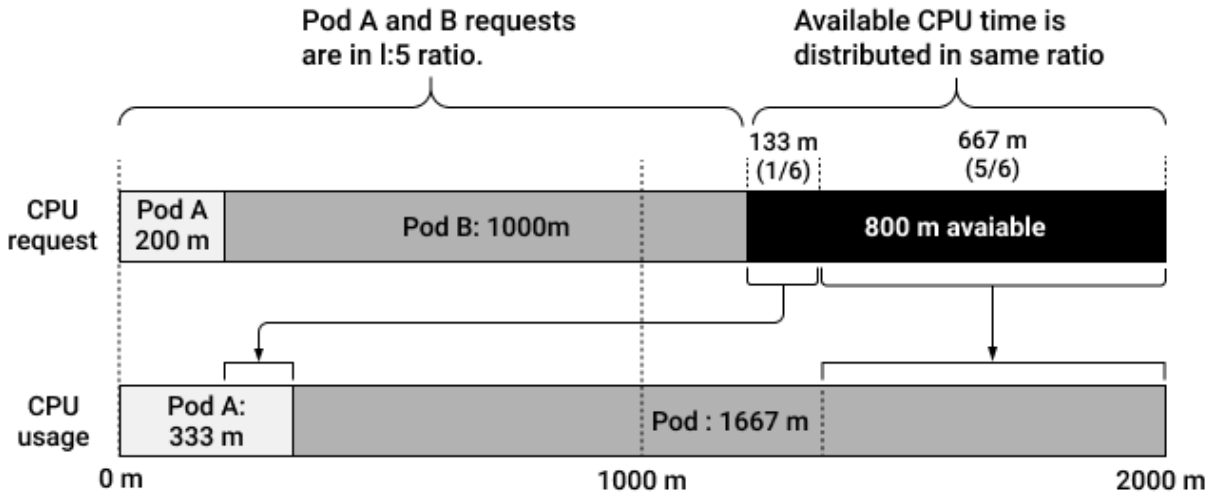
```
kubectl exec -it requests-pod -- /bin/top

Mem: 1572244K used, 163008K free, 1480K shrd, 79780K buff,
817380K cached
CPU: 50.1% usr 49.8% sys  0.0% nic  0.0% idle  0.0% io  0.0% irq
0.0% sirq
Load average: 8.49 3.78 1.58 5/503 38
  PID  PPID USER     STAT   VSZ %VSZ CPU %CPU COMMAND
    1     0 root     R     1296  0.0   0 99.9 dd if /dev/zero of
/dev/null
   34     0 root     R     1304  0.0   0  0.0 /bin/top
```

Это происходит потому, что мы не показали лимиты, и Kubernetes позволил процессу в контейнере использовать все имеющиеся ресурсы. Существует огромная вероятность

возникновения этой ситуации, когда ресурсов будет недостаточно, и мы не сможем больше запускать нагрузку на узле кластера.



Запросы на CPU, в свою очередь, не только решают, где будет запущен pod, но и как будут утилизированы процессы при избытке ресурсов.



Лимиты на ресурсы указываются подобным образом:

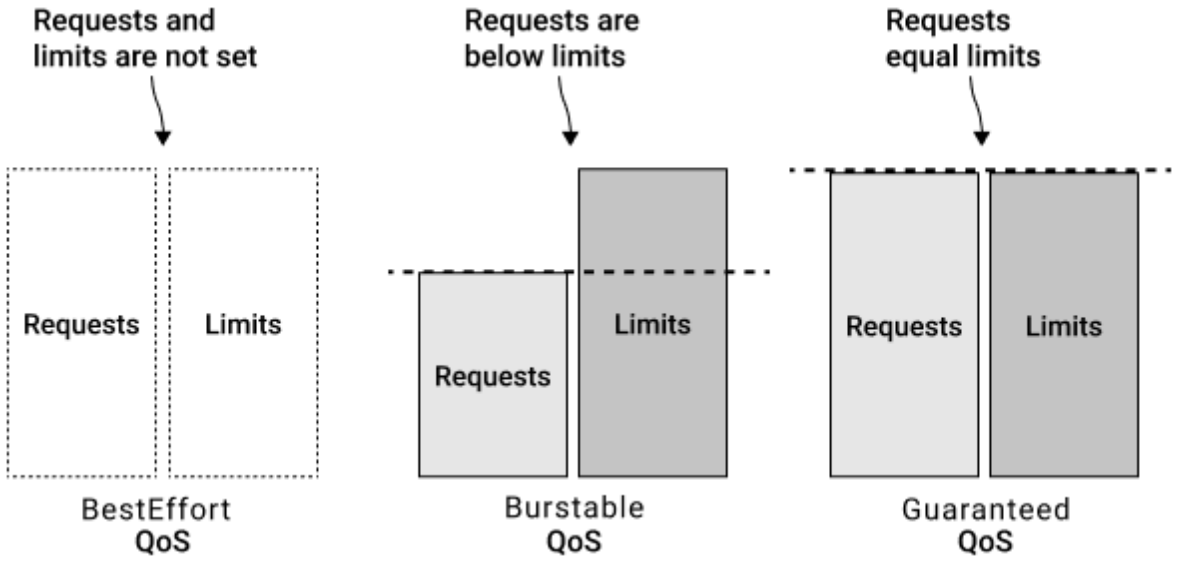
```
apiVersion: v1
kind: Pod
metadata:
  name: requests-pod
spec:
  containers:
  - image: busybox
  command: ["dd", "if=/dev/zero", "of=/dev/null"]
```

```
  name: main
  resources:
    limits:
      cpu: 100m
      memory: 20Mi
```

Если не указать requests, то они будут равны limits. После применения изменения проверяем результат:

```
kubectl exec -it requests-pod -- /bin/top

Mem: 1309420K used, 425832K free, 1480K shrd, 90784K buff,
503788K cached
CPU:  9.1% usr  7.1% sys  0.1% nic 83.4% idle  0.0% io  0.0% irq
0.0% sirq
Load average: 0.16 0.19 0.18 2/494 11
  PID  PPID USER     STAT   VSZ %VSZ CPU %CPU COMMAND
    1     0 root     R     1296  0.0   0  9.9 dd if /dev/zero of
/dev/null
    6     0 root     R     1304  0.0   0  0.0 /bin/top
```

Для более простого управления ресурсами в kubernetes есть QoS классы.
Изначально есть три класса — BestEffort, Burstable, Guaranteed.
Qos класс не задается нигде в описании объекта, он зависит от комбинации requests и limits.
BestEffort — самый низкоприоритетный класс. Он назначается объектам, у которых не указано requests и limits. В самом негативном случае такие объекты могут совсем не получить ресурсов. Они же — первые кандидаты на перемещение на другой узел при перераспределении ресурсов.
Guaranteed — назначается объектам с заданными requests == limits.
Нужно задать эти значения для каждого контейнера, при этом запрошенное и лимитируемое значение cpu и ram должно быть таким же.
Burstable — почти как Guaranteed, только с заданными requests < limits или теми, у которых задан один из параметров.

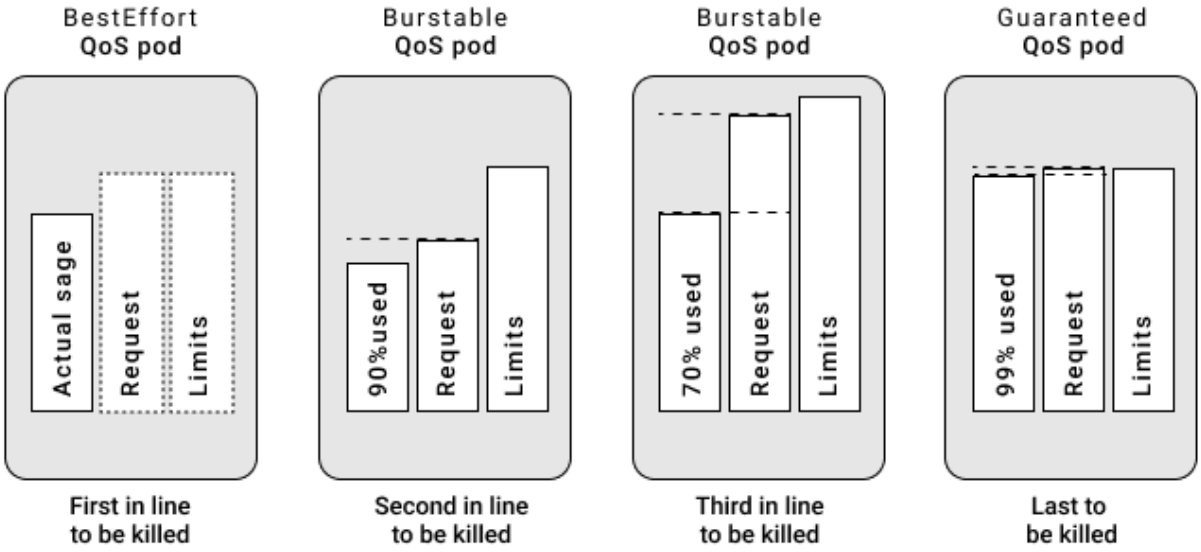


CPU requests vs. limits	Memory requests vs. limits	Container QoS class
None set	None set	BestEffort
None set	Requests < Limits	Burstable
None set	Requests = Limits	Burstable
Requests < Limits	None set	Burstable
Requests < Limits	Requests < Limits	Burstable
Requests < Limits	Requests = Limits	Burstable
Requests = Limits	Requests = Limits	Guaranteed

Если в объекте несколько контейнеров

Container 1 QoS class	Container 2 QoS class	Pods QoS class
BestEffort	BestEffort	BestEffort
BestEffort	Burstable	Burstable
BestEffort	Guaranteed	Burstable
Burstable	Burstable	Burstable
Burstable	Guaranteed	Burstable
Guaranteed	Guaranteed	Guaranteed

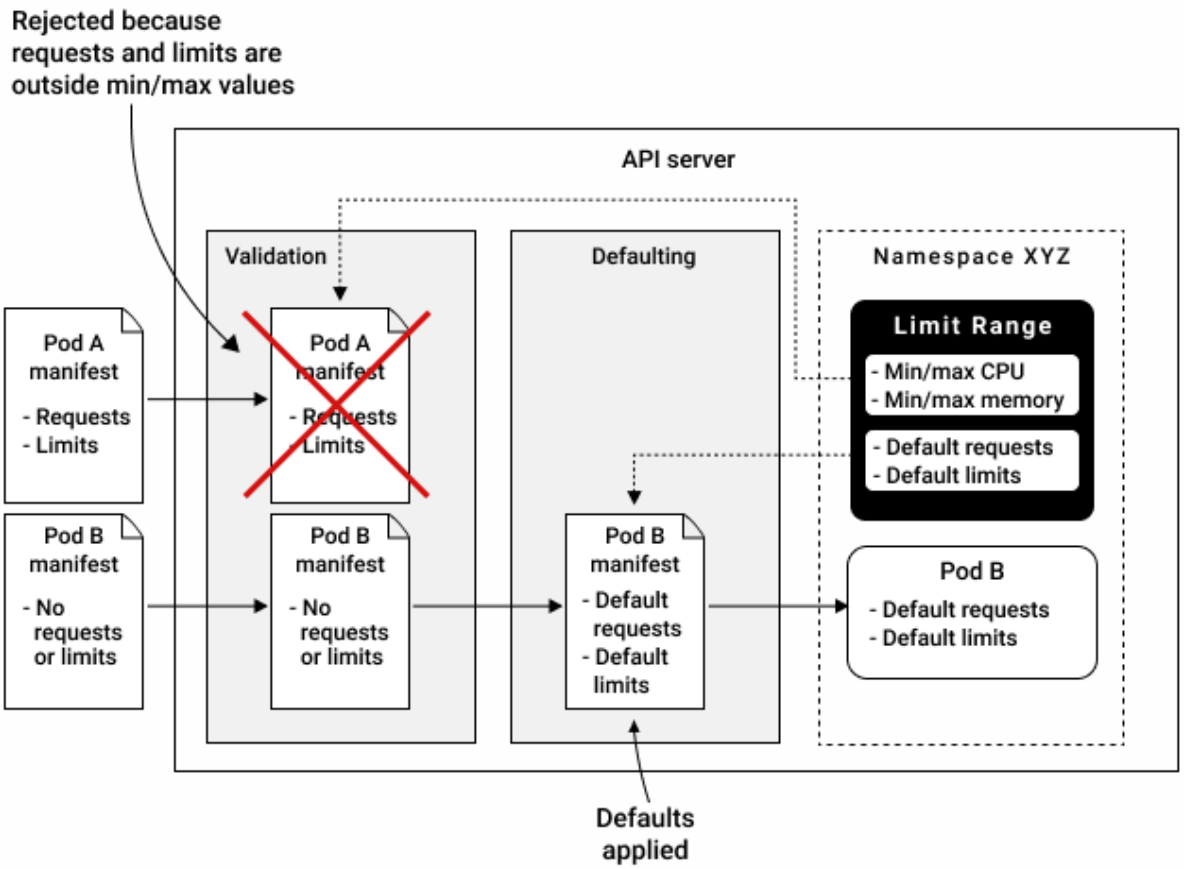
Рассмотрим ситуацию, когда ресурсов недостаточно, например, недостаток памяти. OOM-Killer первыми завершит процессы BestEffort, затем — Burstable и только потом — Guaranteed.



Важно упомянуть, что QoS влияет только на такие ресурсы, как CPU и Memory, но если на сервере не хватает диска для запуска приложения, то несмотря на политики QoS, никакой под на указанной ноде не запустится из-за DiskPressure состояния ноды.
Данные о том, какой QoS выставлен для пода, можно узнать, применив на нем kubectl describe pod ....
LimitRange / ResourceQuota
Для задания лимитов в Kubernetes, кроме уже разобранных ограничений по ресурсам на уровне контейнера, существует еще 2 сущности — LimitRange и ResourceQuota.
Действуют эти сущности на уровне Namespace, а не всего кластера.
LimitRange дает возможность задать рамки и значения по умолчанию для запросов и лимитов на ресурсы, значит, их можно не указывать в контейнерах. Записанные ограничения на уровне конкретных ресурсов имеют больший приоритет.
Пример:

```
apiVersion: v1
kind: LimitRange
metadata:
  name: example
spec:
  limits:
  - type: Pod
    min:
      cpu: 50m
      memory: 5Mi
    max:
      cpu: 1
      memory: 1Gi
  - type: Container
```

```
    defaultRequest:
      cpu: 100m
      memory: 10Mi
    default:
      cpu: 200m
      memory: 100Mi
    min:
      cpu: 50m
      memory: 5Mi
    max:
      cpu: 1
      memory: 1Gi
    maxLimitRequestRatio:
      cpu: 4
      memory: 10
  - type: PersistentVolumeClaim
    min:
      storage: 1Gi
    max:
      storage: 10Gi
```



Таким образом, мы ограничили ресурсы для Pod, контейнеров и PersistenVolumeClaim, а также задали значения по умолчанию для контейнеров.
Для ограничения суммарных ресурсов, разрешенных для использования в Namespace, нужен другой объект — ResourceQuota.
Пример:

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: cpu-and-mem
spec:
  hard:
    requests.cpu: 400m
    requests.memory: 200Mi
    limits.cpu: 600m
    limits.memory: 500Mi
```

Точно также можно задать квоту на PVC, на каждый StorageClass отдельно:

```
apiVersion: v1
kind: ResourceQuota
```

```
metadata:
  name: storage
spec:
  hard:
    requests.storage: 500Gi
    ssd.storageclass.storage.k8s.io/requests.storage: 300Gi
    standard.storageclass.storage.k8s.io/requests.storage: 1Ti
```

И даже ограничить количество объектов в namespace:

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: objects
spec:
  hard:
    pods: 10
    replicationcontrollers: 5
    secrets: 10
    configmaps: 10
    persistentvolumeclaims: 4
    services: 5
    services.loadbalancers: 1
    services.nodeports: 2
    ssd.storageclass.storage.k8s.io/persistentvolumeclaims:
```

Кроме того, ResourceQuota позволяет более гранулярно управлять квотами в рамках QOS класса:

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: besteffort-notterminating-pods
spec:
  scopes:
  - BestEffort
  - NotTerminating
  hard:
    pods: 4
```

Этот пример задает квоту на 4 пода класса BestEffort в состоянии NonTerminating.
То есть, только 4 пода такого QOS-класса могут существовать одновременно в этом namespace.
Pod Disruption Budget

Очень важной темой при работе с сервисами является pod disruption budget. Он позволяет указать Kubernetes, сколько может быть недоступно подов из Deployment, RS или StatefulSet. Допустим, вы хотите вынести ноду из кластера для внутренних работ. Если для сервиса не указан PDB, то Kubernetes просто возьмет и разом начнет переносить все поды, которые относятся к этому деплойменту. Это не очень хорошо, потому что ваш сервис может и не протянуть на оставшихся подах.
Для такой ситуации есть смысл указать PDB. Давайте для начала создадим deployment:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
spec:
  replicas: 5
  selector:
    matchLabels:
      app.kubernetes.io/name: nginx
  template:
    metadata:
      labels:
        app.kubernetes.io/name: nginx
    spec:
      containers:
        - name: nginx
          image: nginx
```

Теперь мы можем создать PDB для этого деплоймента следующего вида:

```
apiVersion: policy/v1beta1
kind: PodDisruptionBudget
metadata:
  name: nginx-pdb
spec:
  minAvailable: 3
  selector:
    matchLabels:
      app.kubernetes.io/name: nginx
```

То есть, мы говорим, что для этого приложения минимальное число доступных подов должно быть 3. Если посмотреть на статус нашей политики:

```
$ kubectl get pdb
NAME        MIN AVAILABLE   MAX UNAVAILABLE   ALLOWED
DISRUPTIONS   AGE
```

```
nginx-pdb   3               N/A               2
39s
```

То можно увидеть, что pdb считает, что можно переносить или убивать 2 пода — поскольку минимально должно быть доступно 3 штуки, а у нас запущено 5 штук.
PDB очень важно настраивать для ваших приложений, поскольку при использовании cordon & drain для отключения нод на maintenance, Kubernetes будет следовать данным правилам для расселения подов.

Полезные ссылки:

- Limit Ranges (official docs)
- Resource Quotas (official docs)
- Quotas and Limit Ranges (openshift official docs)
- Configure Quality of Service for Pods (official docs)
- Everything you Need to Know about Kubernetes Quality of Service (QoS) Classes
- What are Quality of Service (QoS) Classes in Kubernetes
- Managing Compute Resources for Containers (official docs)
- Prevent Downtime with Proper Kubernetes Resource Planning

Задание:

1. Создайте deployment cache-dp с двумя контейнерами memcached, redis.
2. Установите requests / limits таким образом, чтобы QoS для этих подов был Guaranteed.
3. Ограничьте использование ресурсов в namespace default максимально 4 ядрами CPU и 16 Gib памяти.
4. Отправьте задание на проверку.