[KEP 2570] Memory QOS

Created: 2023-01-23
Last updated: 2023-01-23
Status: Draft
Kep-2570
Enhancement Issue
Blog Post

Username	Role	Status	Last Change
David Porter	**Approver**	**PENDING**	**2023-01-23**
Mrunal Patel	**Approver**	**PENDING**	**2023-01-23**
Tim Xu	**Approver**	**PENDING**	**2023-01-23**
Paco Xu	**Approver**	**PENDING**	**2023-01-23**
Sergey Kanzhelev	**Approver**	**PENDING**	**2023-01-23**

Background

In Kubernetes v1.22, an alpha feature named `QOS for Memory' was introduced to improve how Linux nodes implement memory resources requests and limits.

- It uses the memory controller of cgroup v2 to guarantee memory resources in Kubernetes. Following table specifies the mapping of cgroup v2 memory controller's interface files with Kubernetes pod parms:

Cgroup v2	Pod
memory.max	memory.limit
memory.high	MemoryThrottlingFactor * (memory.limit or node allocatable), where memoryThrottlingFactor is 0.8 by default
memory.min	memory.request

- It adds a MemoryThrottlingFactor field in kubelet configuration. 0.8 is set as default memoryThrottling factor.

Note: This feature doesn't use memory.low interface of cgroup v2 memory controller.
TODO(Dixita): It might be worth documenting why we are not using memory.low.

Problem Statement

The current implementation for memory.high is a function of memory.limit or node allocatable if memory.limit is not set. It is based on the following formula:

```
memory.high = MemoryThrottlingFactor * (memory.limit or node allocatable)

The default MemoryThrottlingFactor is set to 0.8.
```

This approach has following problems:

1. **Fails to throttle when requested memory is closer to memory limits (or node allocatable): memory.high < memory.request**
   memory.high is the memory usage level at which throttling occurs. If the requested memory is closer to memory limits, the current formula results in memory.high being less than the memory.request. Since the requested memory is beyond memory throttling limits i.e. memory.high, there won't be any throttling.

   For example,
   memory.request = 850Mi
   MemoryThrottlingFactor = 0.8
   memory.limit = 1000Mi
   Per current implementation,
   memory.high = MemoryThrottlingFactor * memory.limit
   = 0.8 * 1000Mi = 800Mi
   Here, memory.high < memory.request.
   TODO(ndixita): link the code with this check

2. **Throttling is easy to reach and can induce throttling too early**
   A sizable chunk of memory is unused because of early throttling.

   For example,
   - memory.request = 800Mi
     MemoryThrottlingFactor = 0.8
     memory.limit = 1000Mi
     Per current implementation,
     memory.high = MemoryThrottlingFactor * memory.limit
     = 0.8 * 1000Mi = 800Mi
     Throttling occurs at 800Mi, while a sizable chunk of 200Mi is still unused.
   - memory.request = 500Mi
     MemoryThrottlingFactor = 0.6
     memory.limit = 1000Mi
     Per current implementation,
     memory.high = MemoryThrottlingFactor * memory.limit
     = 0.6 * 1000Mi = 600Mi
     Throttling occurs at 600Mi which is just a 100Mi over the requested memory.
     400Mi of memory is still unused as early throttling happens.

3. **Default throttling factor of 0.8 might be too aggressive for applications**
   Some applications have high memory usage closer to memory limits. Throttling will occur when memory consumption reaches 80% of the memory limit as default MemoryThrottlingFactor is 0.8. Such workloads are likely to be always throttled as the memory consumption could be higher than the memory.high throttling level.

   For example, some Java workloads that use 85% of the memory will start to get throttled once this feature is enabled by default. Hence the default 0.8 MemoryThrottlingFactor value may not be a good value.

Limit 1000Mi \ Request factor	**current design: memory.high = MemoryThrottlingFactor * memory.limit (or node allocatable if memory.limit is not set)**
request 0Mi factor 0.6	600Mi (400Mi unused)
request 500Mi factor 0.6	600Mi (early throttling when memory usage is just 100Mi above requested memory; 400Mi unused)
request 800Mi factor 0.6	max (600 < 800 i.e. memory.high < memory.request => no throttling)

request 1Gi factor 0.6	max (600 < 800 i.e. memory.high < memory.request => no throttling)
request 0Mi factor 0.8	800Mi
request 500Mi factor 0.8	800Mi
request 800Mi factor 0.8	max (600 < 800 i.e. memory.high < memory.request => no throttling)
request 500Gi factor 0.4	max(400 < 500 i.e. memory.high < memory.request => no throttling)

Goal

The Memory QOS feature involves behavior change as it makes memory consumption stricter for customer workloads. The workloads might get affected by memory reclamation more now with this change. This document is to have an open discussion around the following topics for smooth change:

1. Alternative formulas to calculate memory.high.
2. Higher default value of MemoryThrottlingFactor.
3. Alternatives for Pod-level Memory QOS configuration.

Alternative Formulas

1. **memory.high as a function of memory.request and memory.limit**

```
memory.high = memory.request + MemoryThrottlingFactor * (memory limit or
node allocatable - memory request)
```

Limit 1000Mi\ Request \ factor	**current design: memory.high**	**Alternative 1 memory.high = memory.request + MemoryThrottlingFactor * memory.limit - memory.request**
request 0Mi factor 0.6	600Mi	600Mi (document: 400Mi is still unused and throttling occurs. It is not recommended to use such a small

		throttling factor to avoid early throttling)
request 500Mi factor 0.6	600Mi	800Mi
request 800Mi factor 0.6	max (not throttling)	920Mi
request 1Gi factor 0.6	max	max
request 0Mi factor 0.8	800Mi	800Mi
request 500Mi factor 0.8	800Mi	900Mi
request 800Mi factor 0.8	max	960Mi
request 500Mi factor 0.4	max	700Mi
request 996Mi factor 0.6	max(no throttling)	998.4 (Huge page size: say 2Mi, throttling should have occurred at 998Mi. Hence page size needs to be accounted for in the formula)
request 996Mi Factor 0.8	max (no throttling)	999.2 (Huge page size: say 2Mi, throttling should have occurred at 998Mi. Memory less than page size won't be allocatable Hence page size needs to be accounted for in the formula.)

Pros:

- memory.high is never less than memory.request.
  - As, memory.high = memory.request + MemoryThrottlingFactor * (memory.limit - memory.request), memory.high >= memory.request

Cons:

- Formula is a little more complicated for customers than the current implementation.
- Early throttling can still occur when the MemoryThrottlingFactor is set to a low value.

Note: The cons can be addressed by providing proper guidance around recommended values of MemoryThrottlingFactor.

2. **[Preferred Alternative] memory.high as a function of memory.request, memory.limit and page size.**
   This is an improvement to formula (1) that makes sure memory.high is divisible by pageSize since memory is always requested in chunks of pageSize.

   Formula:
   ```
   memory.high = floor(memory.request + throttling factor * (memory limit or
   node allocatable - memory request)/pageSize) * pageSize
   ```

   Cons:
   - Formula is a little complicated for customers who do not want to use default MemoryThrottlingFactor
   - Early throttling for small MemoryThrottlingFactor

   Note: The cons can be addressed by providing proper guidance around recommended values of MemoryThrottlingFactor.

3. **Set default MemoryThrottlingFactor to a high value 0.9 or 0.95 with the current implementation and make it configurable.**

   Pros:
   - Simple formula.

   Cons:
   - Failure to throttle as memory.high can be lesser than memory.request.

     For example, memory.request = 980Mi
     memory.limit = 1000Mi
     memory.high = 0.95 * 1000Mi = 950Mi
     Since memory.high < memory.request, throttling won't occur

4. **Pod Level setting for a soft memory request**
   For example, custom setting in the POD YAML file to throttle at 180Mi.

```
apiVersion: v1
kind: Pod
metadata:
  name: example
spec:
  containers:
  - name: nginx
    resources:
      requests:
        memory: "200Mi"
        cpu: "250m"
```

```
      throttlingLimits:
        memory: "180Mi"
      limits:
        memory: "64Mi"
        cpu: "500m"
```
Cons:

- Requires changing the APIs to add a new field throttlingLimits.
- Understanding the throttlingLimits can be tricky for customers. Percentage of memory usage through setting defaultMemoryThrottling factor is easier to gauge.

Default MemoryThrottlingFactor

With 0.8 as default MemoryThrottlingFactor, 20% of the memory will remain unused. It might be worth considering a higher value of MemoryThrottlingFactor.

The table below runs over the examples with different values using the formula:

```
memory.high = memory.request + MemoryThrottlingFactor * (memory.limit -
memory.request
```

Limit 1000Mi	**throttlingMemoryFactor 0.6**	**throttlingMemory Factor 0.8**	**throttlingMemory Factor 0.9**	**throttlingMemory Factor 0.95**
request 0Mi	600Mi	800Mi	900Mi	950Mi
request 100Mi	640Mi	820Mi	910Mi	955Mi
request 200 Mi	680Mi	840Mi	920Mi	960Mi
request 300 Mi	720Mi	860Mi	930Mi	965Mi
request 400 Mi	760Mi	880Mi	940Mi	970Mi

request 500Mi	800Mi	900Mi	950Mi	975Mi
request 600Mi	840Mi	920Mi	960Mi	980Mi
request 700Mi	880Mi	940Mi	970Mi	985Mi
request 800Mi	920Mi	960Mi	960Mi	990Mi
request 900Mi	960Mi	980Mi	980Mi	995Mi
request 1000Mi	1000Mi	100Mi	1000Mi	1000Mi

Higher values of 0.9, 0.95 and so on cause less aggressive memory throttling limits, and results in lesser unused memory. While very high values of 0.95 or a greater factor might not be as effective as in this case memory.high could be almost equal to memory.limit.

[Actionable] Discussion for readers
What should we pick as the default value of MemoryThrottlingFactor? 0.9 or 0.95
Author's opinion: 0.9 can be set as an initial default value, and can be changed to a higher value of 0.95 based on the feedback from customers.

Alternatives For Pod-Level Memory QOS Configuration

MemoryThrottlingFactor is a kubelet configuration that gets applied to nodes. Following are the alternatives for implementing Memory QOS configuration at Pod level:

1. **[Preferred] QOS Classes**
    - Continue allowing customers to MemoryThrottlingFactor at node level.
    - Pods should comply with the QOS classes. Memory QOS behavior for the QOS classes is as follows:

i. Guaranteed class
The pod gets a Guaranteed class if the request and limit values are the same. Guaranteed class pods are the highest priority pods and we disable throttling for these pods by setting memory.high equal to memory.limit.

```
memory.high = memory.limit = memory.request
```

ii. Burstable
The pod gets a Burstable class if the requested memory is lower than the limit.

Case 1: When memory.request and memory.limit are set

```
memory.high = floor [ (memory.request + memoryThrottlingFactor
* (memory.limit - memory.request)) / pageSize ] * pageSize
```

Case 2. When memory.request is set, memory.limit is not set

```
memory.high = floor[ (memory.request + memoryThrottlingFactor *
(node allocatable - memory.request) / pageSize) * pageSize
```

Case 3. When memory.request is not, memory.limit is set

```
memory.high = floor[ (memoryThrottlingFactor * memory.limit) /
pageSize) * pageSize
```

iii. BestEffort
The pod gets a BestEffort class if memory.limit and memory.request are not set.

```
memory.high = floor[ (memoryThrottlingFactor * node
allocatable) / pageSize) * pageSize
```

Pros

- Memory QOS complies with QOS which is a wider known concept.
- Simpler to understand as it requires setting only 1 kubelet configuration.

Cons

- There might be customers who require setting different throttling factors per pod. Based on feedback from customers, we could decide if alternative solutions cover more customer use cases in future versions.

2. **Allow customers to set MemoryThrottlingFactor for each pod in annotations.**
    - Add a new annotation for customers to set memoryThrottlingFactor to override kubelet level MemoryThrottlingFactor.

    Pros
    - Allows more flexibility.
    - Can be quickly implemented.

    Cons
    - Customers might not need per pod memoryThrottlingFactor configuration.
    - TODO(ndixita):t not good from API side

3. **Allow customers to set MemoryThrottlingFactor in pod yaml**
    - Add a new field in API for customers to set memoryThrottlingFactor to override kubelet level MemoryThrottlingFactor.

    Pros
    - Allows more flexibility.

    Cons
    - Customers might not need per pod memoryThrottlingFactor configuration.
    - API changes take a lot of time, and we might eventually realize that the customers don't need per pod level setting.

[Actionable] Discussion for readers
How should we allow pod level configuration?
Author's opinion from meetings discussions: Start with QOS classes, and gather feedback. Based on feedback from customers, decide if alternative solutions make more sense.

Code Changes

TODO(ndixita): Add code snippets