Vertical Scaling Proposal

Last Modified: October 20, 2017

Status: Shared with Kubernetes Resource Management Workgroup

Authors:

Karthick Rajamani (karthick@us.ibm.com)

Youngjae Lee (leeyo@us.ibm.com)

Alexandre Ferreira (apferrei@us.ibm.com)

Abstract

This proposal is to enable the resizing of resources allocated to running containers and pods without having to restart the pods. The initial focus is on enabling this for Statefulsets, in particular, though this may be use-able for stand-alone/controller-free pods and Deployments as well.

Motivation

Resources needed by containers and pods can change over time for a variety of reasons - moving from live-test mode to production usage, change in user load or dataset sizes each of which again might come about for a variety of reasons. Deployments and Statefulsets (with v 1.7) support the capability to change *Request* and *Limit* values specified for container through supported pod spec update methods. However, they currently require the pods be restarted to run with the new resource sizes.

There are a few reasons why restarting may not be desirable particularly for stateful services.

1) Moving or copying existing state information or re-sharding employing horizontal scaling may be expensive or even infeasible depending on the service or underlying application implementation.

2) Services with large amount of state associated with specific instances may encounter temporary loss in access to the specific state information while its corresponding container is being restarted. Or they might encounter non-trivial delays in getting back to expected performance.
3) State may be maintained on node-local storage/devices and restart might place the container/pod on a different node (unless suitably constrained). Might create more fragile scaling option for service and/or additional burden (to constrain placement) on user.
4) Where the restart is not needed, unnecessary load may be created on Kubernetes components that are responsible for pod deletion and creation potentially impacting Kubernetes scalability. Resizing running containers (and their pods) in place could avoid that additional load.

Conversely, resizing without restart may not be the right option for all applications and the right option may also be resource dependent. For example, an application might (or have the means to) assess resources available to it only at its initialization time and be unable to adjust its usage if those are dynamically changed. Or it might be able to adjust to changes in certain resources (most can for CPU) but not for others (some application have particular difficulty freeing memory they have taken up). So we need the means to express and implement the best approach for resizing for each workload and resource.

API and Usage

To express the policy for resizing we plan to introduce resource attribute *resourceResizePolicy* with following choices for value:

- **RestartOnly** (default),
- **LiveResizeable.**
- Default for pod types not supporting resizing would be None/NoResize.

This attribute will be available per resource (such as cpu, memory) and so is adequate to indicate whether the workload can handle and prefer a change in each resource's allocation for it without restarting.

With potentially multiple containers and multiple resizeable resources for each in a Pod, the response to an update of the pod spec will be determined by the a precedence order among the attribute values with **RestartOnly** dominating **LiveResizeable,** i.e., if two resources have been resized in the update to the spec and one of them has a policy of RestartOnly then the pod would be restarted to realize both updates.

We can optionally introduce an *action* annotation resourceResizeAction with following choices for value:

- **Restart** (default),
- **LiveResize**,

- **LiveResizePreferred**

If used, this would be included as part of the patch or appropriate update command providing the spec update for the resize. It would indicate the preference of user at the time of resize. Specifically, **Restart** for resourceResizeAction would indicate the pod be restarted for the corresponding resizing of resource(s), **LiveResize** would indicate the pod not be restarted the resize be realized live, and **LiveResizePreferred** would indicate that the resize be realized preferrably live but if that fails for any reason to accomplish it with a restart.

```
Resources:
        Requests:
                CPU: 100m
                Memory: 1Gi
        Limits:
                CPU: 1000m
                Memory: 1Gi
        resourceResizePolicy:
                CPU: LiveResizeable
                Memory: RestartOnly
```

Example 1: Usage of resourceResizePolicy attribute in Pod spec

For Example 1, if there is a change to *cpu* request or limit it can be vertically scaled only if the *memory* request and limit remained the same, otherwise the RestartOnly policy for *memory* would override the policy for CPU, and the Pod (container, if container-alone restart is allowed) would need to be restarted.

Resources:
Requests:
CPU: **100m**
Memory: 1Gi
Limits:
CPU: 1000m
Memory: 1Gi
resourceResizePolicy:
CPU: LiveResizeable
Memory: RestartOnly
	“annotations” : {
	“resourceResizeAction”: “ResizePreferred”
	}
	
	Resources:
	Requests:
	CPU: **400m**
	Memory: 1Gi
	Limits:
	CPU: 1000m
	Memory: 1Gi
	resourceResizePolicy:
	CPU: LiveResizeable
	Memory: RestartOnly

Example 2: Change in CPU resource *Request* size and usage of resourceResizeAction annotation with the updated spec.

Combining Stateful Set Update options with Vertical Scaling

With Kubernetes v1.7 Stateful sets support two update options with spec.UpdateStrategy.type, where OnDelete applies the changed spec when restarting a pod after an explicit delete command and the RollingUpdate is applied by the controller by restarting each of the pods that are members of the Stateful set with the changed values, in reverse ordinal sequence (see https://kubernetes.io/docs/tutorials/stateful-application/basic-stateful-set/#updating-statefulsets for more details).

In the table below we propose how the resource resizing directives for vertical scaling can be applied in conjunction with the Stateful Set update strategy introduced in v1.7.

resourceResizePolicy (resourceResizeAction)	OnDelete (default in v1.7)	RollingUpdate
RestartOnly (**or** Action=Restart)	Resize resource with restart (with delete command)	Resize resource with restart (with allowed spec update commands)

LiveResizeable (**and** Action=LiveResize)	Resize resource live only (with allowed spec update commands)	Resize resource live only (with allowed spec update commands)
LiveResizeable (**and** Action=LiveResizePreferred **or** no Action specified)	Resize resource with live resize preferred (with allowed spec update commands), i.e., resize resource with restart (with delete command) if not able to resize live.	Resize resource with live resize preferred (with allowed spec update commands), i.e., resize resource with restart (managed by controller) if not able to resize live.

Desired Approach

Any valid method to update the pod spec should be applicable for vertical scaling, e.g., using kubectl commands *set, patch, apply, edit.* Logs associated with the pod will capture the failure/success of the resize command. The controller will continue to attempt the update to the spec while there is a difference between the current size and size in updated spec. If an update is partially successful, user can know this from the logs and attempt to rectify the situation by submitting new updates (that can restore original size(s) or go for a size feasible on all the nodes).

The policy for vertical scaling can itself be mutable. It is desirable to submit a change in policy, confirm its acceptance (by re-reading the changed pod spec) and then submit a change to a resource size impacted by that policy.

Design and Implementation

This section highlights some of implementation details by describing API changes and the workflow of vertical scaling. For now, this covers only the part of pod-level vertical scaling. It will be updated about StatefulSet later.

1. Changes on API and key components

This section describes briefly API changes for pod-level vertical scaling and the related changes on the API server, the Scheduler, and the Kubelet.

- ResizeRequest

A new data structure, ResizeRequest, is added to v1.PodSpec:

```
const (
   ResizeRequested ResizeStatus = "Requested"
   ResizeAccepted ResizeStatus = “Accepted”
   ResizeRejected ResizeStatus = “Rejected”
   ResizeNone      ResizeStatus = "None"
)

type ResizeRequest struct {
   RequestStatus ResizeStatus
   NewResources  []ResourceRequirements // indexed by containers’ index
}

Type PodSpec {
   …
   ResizeRequest ResizeRequest
   ...
}
```

ResizeRequest has two variables, RequestStatus and NewResources. RequestStatus represents the status of a resource resizing request. ResizeRequested indicates resource resizing for a pod is requested. ResizedAccepted and ResizedRejected means that the requested resource resizing is accepted and rejected, respectively, by the Scheduler. The NewResources is an array indexed by a container’s index and its each entry holds new resource requirements of a container that needs to resize.

Given a new PodSpec with new resource requirements from a client, first the API server validates it. If it is valid, the API server sets the RequestStatus to ResizedRequested and copies the new resource requirements of each container into the NewResources. Also, the API server restores the resource requirements of each container of the PodSpec to the original and writes the revised PodSpec to ETCD to communicate with the Scheduler. This is because at this moment the PodSpec on ETCD shouldn’t be updated with new resource requirements.

For a pod with ResizeRequested, the Scheduler checks if the node on which the pod currently runs has enough resources to resize the pod. If the resource resizing is feasible, the Scheduler sets the RequestStatus of the pod to ResizedAccepted and issues a Resizing API operation, which will be described below, to the API server. Otherwise, the Scheduler sets the RequestStatus to ResizedRejected and issues a Resizing API operation.

- Resizing

A new API, Resizing, for the scheduler is introduced:

```
// Resizing resizes the resources allocated to a pod
```

```
type Resizing struct {
    metav1.TypeMeta
    metav1.ObjectMeta
    Request ResizeRequest
}
```

Resizing has the metadata of a pod to resize and a value of ResizeRequest that holds the status of a resizing request, which indicates whether the resizing is feasible or not, and new resource requirements of the pod.

Once the Scheduler determine that a resource resizing on a pod is feasible, or not, it notifies to the API server via this Resizing API. ObjectMeta.Namespace/Name are set to the Namespace and Name of the pod to resize and ResizeRequest is set with new resource requirements.

Given a Resizing API operation, the ResizeStatus of the PodSpec of a Pod is updated according to that of the Resizing operation. If the ResizeStatus is ResizeAccepted, the API server updates the ResourceRequirement of each container of a pod with new resource requirements on ETCD.

- PodResized

A new pod condition, PodResized, and condition status for that is added to v1.PodCondition:

```
// These are valid conditions of pod.
const (
       // PodResized represents the status of the resizing process for this pod
    PodResized PodConditionType = "PodResized"
    PodReasonUnresizable   = "Unresizable"
    PodReasonResizerFailed = "ResizerFailed"
)
const (
    ConditionRequested ConditionStatus = "Requested"
    ConditionAccepted  ConditionStatus = "Accepted"
    ConditionRejected  ConditionStatus = "Rejected"
    ConditionDone      ConditionStatus = "Done"
)
```

The pod condition of PodResized represents the status of the resizing process. The PodResized Condition is updated by the Kubelet according to the ResizeStatus.

Basically, when the ResizeStatus is changed, the Kubelet updates the PodResized condition accordingly. In case of ConditionDone, the Kubelet sets the PodResized of a Pod to it when all the containers that need to be resized complete to be resized.

- UpdateContainer -> changes to leverage the UpdateContainerResource newly added at v1.8

A new CRI interface, UpdateContainer, is introduced:

```
type LinuxContainerResources struct {
    CpuPeriod          int64
    CpuQuota           int64
    CpuShares          int64
    MemoryLimitInBytes int64
    OomScoreAdj        int64
}

service RuntimeService {
    ...
    rpc UpdateContainer(UpdateContainerRequest) returns (UpdateContainerResponse) {}
    ....
}

message UpdateContainerRequest {
    string container_id = 1;
    LinuxContainerResources resources = 2;
}

message UpdateContainerResponse {}
```

If a pod is updated with new resource requirements and it is live resizable, the Kubelet updates the cgroup configuration of containers of the pod with UpdateContainerResources CRI interface. If it is not live-resizable for some reasons (e.g. a resizePolicy is RestartOnly) , the Kubelet just kills the existing container and recreate a new container with new resource requirements, like the case where the container's Image is updated.

- A new additional hash added for Kubelet to detect a change on resource requirements

In order to watch resource requirement changes efficiently, a new additional hash is added to kubecontainer.ContainerStatus (and that is also stored as a one of the container's labels). This hash is calculated with a container's spec munged with an empty v1.ResourceRequirements. In addition to the existing container spec's hash, Kubelet uses this new hash to detect a change on resource requirements of a container. Without this new hash, still Kubelet could detect a change

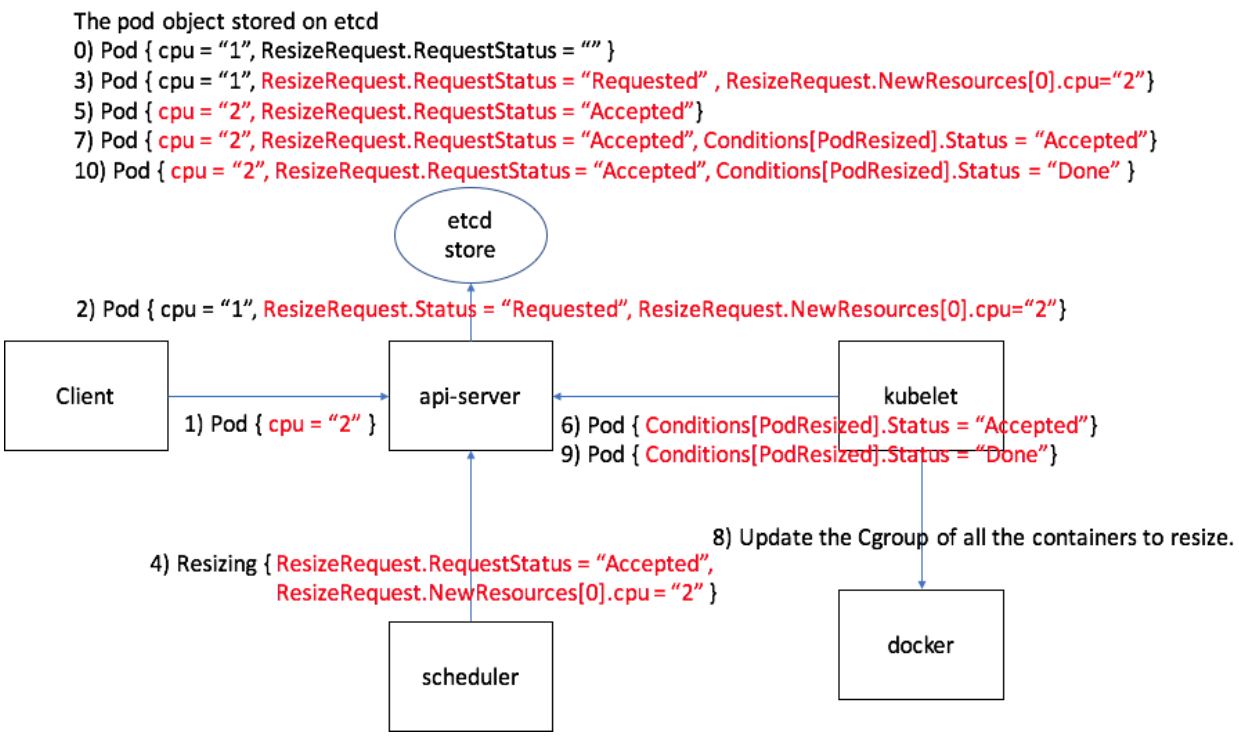
on resource requirement with the existing hash, but needs to compare every entry in the container's spec to identify which entry changes. The following is the details.

```
expectedHash := kubecontainer.HashContainer(&container)
containerChanged := containerStatus.Hash != expectedHash
if containerChanged {
    // Something in the container's spec changed.
    // So, see if it is just a change only on resource requirements.
    mungedContainer := container
    mungedContainer.Resources = v1.ResourceRequirements{}
    expectedHashNoResources := kubecontainer.HashContainer(&mungedContainer)
    containerChangedToStart = containerStatus.HashNoResources !=
expectedHashNoResources

    if containerChangedToStart {
        // This is a change on something other than resource requirements, so it needs to restart
a container
    } else {
        // This is a change only on resource requirements.
    }
}
```

2. Workflow

This describes the sequence of a pod-level vertical scaling example to resize the CPU resource of a pod from 1 to 2.



0) A pod has 1 CPU.
1) A client requests resource resizing on the pod with a new PodSpec with 2 CPU.
2) The API server updates the PodSpec with ResizeRequest on *etcd*
3) The PodSpec is updated on *etcd*.
4) The Scheduler checks if the resizing is feasible and, if so, issues a Resizing API operation with the ResizeRequest of the "Accepted" status.
5) The API server updates the PodSpec with the new resource requirement on *etcd* and also modifies the ResizeRequest of the PodSpec to "Accepted".
6), 7) The Kubelet updates the PodResized condition to "Accepted".
8) The Kubelet detects the change of resource requirements on the container and updates the cgroup configuration of the container via UpdateContainerResources CRI interface.
9) The Kubelet modifies the status of the PodResized condition to Done after every update on the cgroup configuration of all containers to resize is done.
10) It completes to resize the pod to have 2 CPUs.

3. Implementation Phases

Phase 1 - Introduce live and in-place vertical scaling at pod-level
Status: **Done**, a working prototype implemented originally in v1.6 and ported to v1.7 and master (v1.8-alpha)

The working code is available at https://github.com/YoungjaeLee/kubernetes (the qos-master branch)

Phase 2 - Adding the pod-level vertical scaling in StatefulSet
Status: being implemented in master (v1.8-alpha)

Related issues

1. QoS class change by resize is not supported.

For each of the Burstable and the Best-effort class, the Kubelet maintains a class-level cgroup under the ‘kubepods’ cgroup, which is the parent cgroup of pods of each QoS class. (e.g. kubepods/burstable is the parent cgroup of Burstable pods). So, in order to change the QoS class of a pod, it needs not only to resize resources, but also to change its parent cgroup properly. But, once a pod is created, its parent cgroup cannot be changed with the current Docker API. So, the QoS-class of a pod cannot be changed by resource resizing.

2. Memory-resizing to change a request value might not take effect for Burstable pods.

For Burstable pods, a request value for memory resource determines the value of a score for the OOM killer, but Docker doesn’t support to change dynamically the score of an existing container. So, the change to a memory request value doesn’t take effect for Burstable pods on the OOM killer’s behavior. But, for Guaranteed and Best-effort pods, this is not an issue because the score is fixed regardless of its memory request value. (in this case, the memory request value is used only for admission control by Scheduler)

3. Memory-resizing to decrease its limit may fail on the Kubelet in some circumstances.

By default, the Kubelet requires disabling swap. With swap disabled, memory-resizing to decrease fails when there is not enough free memory that can be reclaimed. Specifically, the cgroup change to write a new limit value to memory.limit_in_bytes that is smaller than the current value fails as the Linux kernel fails to reclaim memory in use exceeding the new value.

4. Memory-resizing on memory-backed storage (emptydir backed by memory)

In the perspective of resizing, there is no difference between normal memory and the emptydir memory. Basically, if the memory allocated to a container is resized, the amount of memory available to an emptydir changes accordingly.

With respect to usage accounting, the emptydir memory has a little difference (not specific to resizing.). The emptydir memory is accounted to a container that allocates the memory to store a file in the emptydir. For example, in a case where two containers share a memory-backed emptydir, the memory used to store a file is accounted to one of the containers that created the file and wrote its data, even though the other container is able to read/modify the file as if the file resides on its own allocated memory. If the second container appends some data at the end of

the file, the corresponding memory is accounted to the second container since it is allocated by the second container. This is confirmed by the following simple experiment with two containers running in a single pod sharing a memory-backed emptydir.

- Runs a pod of two ubuntu containers sharing an emptydir “/emptydir” backed by memory.

```
root@arly065:~/yaml/emptydir# cat emptydir-pod.yaml
apiVersion: v1
kind: Pod
metadata:
   name: ubuntu
spec:
   restartPolicy: Always
   containers:
   - name: ubuntu
     image: ubuntu:16.04
     command: ["/bin/bash"]
     stdin: true
     tty: true
     resources:
       limits:
          cpu: "12"
          memory: "2Gi"
       requests:
          cpu: "2"
          memory: "1Gi"
       resizePolicy:
          cpu: "LiveResizable"
          memory: "LiveResizable"
     volumeMounts:
     - name: mem-emptydir
       mountPath: /emptydir
   - name: dummy
     image: ubuntu:14.04
     command: ["/bin/bash"]
     stdin: true
     tty: true
     resources:
       limits:
          cpu: "12"
          memory: "2Gi"
       requests:
          cpu: "2"
          memory: "1Gi"
       resizePolicy:
```

```
          cpu: "LiveResizable"
          memory: "LiveResizable"
      volumeMounts:
      - name: mem-emptydir
        mountPath: /emptydir
    volumes:
      - name: mem-emptydir
        emptyDir:
          medium: "Memory"
root@arly065:~/yaml/emptydir# kubectl create -f emptydir-pod.yaml
pod "ubuntu" created
root@arly065:~/yaml/emptydir# kubectl get pods
NAME      READY     STATUS    RESTARTS   AGE
ubuntu    2/2       Running   0          4s
```

- On one container, creates a 512M file with dd in the emptydir and its memory usage accounted to the container

```
root@arly065:~/yaml/emptydir# kubectl attach -it ubuntu -c ubuntu
If you don't see a command prompt, try pressing enter.
root@ubuntu:/#
root@ubuntu:/# cd /emptydir/
root@ubuntu:/emptydir# dd if=/dev/zero of=data bs=1M count=512
512+0 records in
512+0 records out
536870912 bytes (537 MB, 512 MiB) copied, 0.254545 s, 2.1 GB/s
root@ubuntu:/emptydir# cat /sys/fs/cgroup/memory/memory.stat
cache 536965120 // the memory space for “data” file is accounted.
rss 602112
rss_huge 0
mapped_file 0
dirty 0
writeback 0
pgpgin 131849
pgpgout 607
pgfault 1228
pgmajfault 1
inactive_anon 536879104
active_anon 581632
inactive_file 73728
active_file 0
unevictable 0
hierarchical_memory_limit 2147483648
total_cache 536965120
```

```
total_rss 602112
total_rss_huge 0
total_mapped_file 0
total_dirty 0
total_writeback 0
total_pgpgin 131849
total_pgpgout 607
total_pgfault 1228
total_pgmajfault 1
total_inactive_anon 536879104
total_active_anon 581632
total_inactive_file 73728
total_active_file 0
total_unevictable 0
root@ubuntu:/emptydir# ls -la
total 524292
drwxrwxrwt  2 root root        60 Oct 20 20:07 .
drwxr-xr-x 36 root root      4096 Oct 20 19:10 ..
-rw-r--r--  1 root root 536870912 Oct 20 20:07 data
root@ubuntu:/emptydir#
```

- On the other container, be able to read/write the file without any accounting change.

```
root@arly065:~/yaml/emptydir# kubectl attach -it ubuntu -c dummy
If you don't see a command prompt, try pressing enter.
root@ubuntu:/#
root@ubuntu:/#
root@ubuntu:/# cd /emptydir/
root@ubuntu:/emptydir# ls -la
total 524292
drwxrwxrwt  2 root root        60 Oct 20 20:07 .
drwxr-xr-x 36 root root      4096 Oct 20 19:10 ..
-rw-r--r--  1 root root 536870912 Oct 20 20:07 data
root@ubuntu:/emptydir# dd if=data of=/dev/null bs=1M count=512
512+0 records in
512+0 records out
536870912 bytes (537 MB) copied, 0.153266 s, 3.5 GB/s // can read the file at memory speed.
root@ubuntu:/emptydir# dd if=/dev/zero of=data bs=1M count=256 conv=notrunc
256+0 records in
256+0 records out
268435456 bytes (268 MB) copied, 0.113188 s, 2.4 GB/s // can write the file at memory speed
root@ubuntu:/emptydir# cat /sys/fs/cgroup/memory/memory.stat
```

cache 684032 // the memory for “data” file is not counted.
rss 626688
rss_huge 0
mapped_file 45056
dirty 0
writeback 0
pgpgin 1477
pgpgout 1157
pgfault 2292
pgmajfault 9
inactive_anon 4096
active_anon 598016
inactive_file 585728
active_file 77824
unevictable 0
hierarchical_memory_limit 2147483648
total_cache 684032
total_rss 626688
total_rss_huge 0
total_mapped_file 45056
total_dirty 0
total_writeback 0
total_pgpgin 1477
total_pgpgout 1157
total_pgfault 2292
total_pgmajfault 9
total_inactive_anon 4096
total_active_anon 598016
total_inactive_file 585728
total_active_file 77824
total_unevictable 0
root@ubuntu:/emptydir#

- If the other container appends data at the end of the file, the additional memory usage is accounted to the other container.

root@ubuntu:/emptydir# **dd if=/dev/zero bs=1M count=256 conv=notrunc >> data // appends 256MB to the end of the file**
256+0 records in
256+0 records out
268435456 bytes (268 MB) copied, 0.147038 s, 1.8 GB/s
root@ubuntu:/emptydir# ls -la
total 786436
drwxrwxrwt 2 root root   60 Oct 20 20:07 .

drwxr-xr-x 36 root root 4096 Oct 20 19:10 ..
-rw-r--r-- 1 root root 805306368 Oct 20 20:11 data
root@ubuntu:/emptydir# cat /sys/fs/cgroup/memory/memory.stat
cache 269119488 // the memory space for 256MB data appended is accounted to this container.
rss 610304
rss_huge 0
mapped_file 45056
dirty 0
writeback 0
pgpgin 67449
pgpgout 1597
pgfault 3012
pgmajfault 9
inactive_anon 268439552
active_anon 585728
inactive_file 266240
active_file 397312
unevictable 0
hierarchical_memory_limit 2147483648
total_cache 269119488
total_rss 610304
total_rss_huge 0
total_mapped_file 45056
total_dirty 0
total_writeback 0
total_pgpgin 67449
total_pgpgout 1597
total_pgfault 3012
total_pgmajfault 9
total_inactive_anon 268439552
total_active_anon 585728
total_inactive_file 266240
total_active_file 397312
total_unevictable 0
root@ubuntu:/emptydir#

5. To be added as raised.