Note: This is a public document

Design: Task Resource Limits

Authors	Greg Mann <greg@mesosphere.io>
Revision	1.1
Status	Draft

JIRA: [MESOS-10001]

Introduction

User Impact Summary

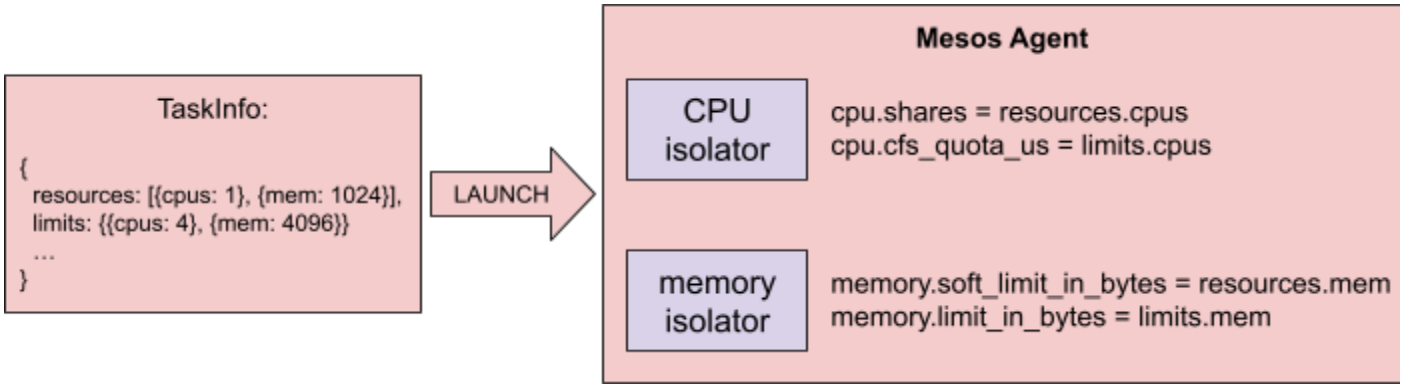
Operators

- Operators will be able to inspect tasks' resource limits via the Mesos UI and operator API

Frameworks

- Frameworks will be able to specify resource limits for tasks when submitting 'TaskInfo'
- Frameworks will be able to specify the level of isolation they desire when launching task groups - CPU and memory may be isolated at the executor container level, or the task container level.

Architecture Overview



Motivation

1. Provide better support for tasks which exhibit a resource spike upon first launch, followed by decreased resource requirements. This is especially an issue for memory,

where exceeding the resource allocation can have a more extreme effect on the task, but could apply to CPU as well.
2. Increase overall resource utilization by allowing tasks which could make use of unused CPU or memory to do so.

Related Implementations

Kubernetes

API

Operators can set resource “request” and “limit” on a container in a pod specification:

```
   resources:
      requests:
        memory: "64Mi"
        cpu: "250m"
      limits:
        memory: "128Mi"
        cpu: "500m"
```

The “request” is used for pod scheduling, and determines the minimum resources that must be kept available for that container.

The “limit” is used while the pod is running to set a hard limit on the resource consumption of the pod.

Goals

- Enable tasks and task groups to “burst” and take advantage of CPU and memory beyond their bare-minimum requirement, when available on the agent
- Enable schedulers to specify a hard limit of CPU and memory, above which their task/task group will never be allowed to consume
- Update the Linux CPU and memory isolators to create nested cgroups for nested containers so that CPU and memory requests/limits are enforced on a per-nested-container basis; design doc for this work can be found at this link.

Non-Goals

- Allow schedulers to run “best-effort” tasks which request no minimum resources and are allowed to “burst” into available CPU/memory headroom on the agent. This is kept as future work in order to reduce the scope of this design.
- Allow schedulers to specify resource limits for executors

Proposed API

The resource “request” for CPU and memory of a task will be defined as the sum of all CPU and memory resources present in the ‘resources’ field in ‘TaskInfo’. The CPU/memory request of a task is the amount of CPU/memory which is guaranteed to be available to the task at all times.

The ‘TaskInfo’ message will be extended with a new ‘limits’ field which allows schedulers to specify resource “limits” on their tasks:

- The ‘limits’ map will be restricted so that the only allowed keys are “cpus” and “mem”. The resource limit for a task is the maximum amount of each resource that the task will be permitted to consume.

```
message TaskInfo
{
  . . .

  map<string, Value.Scalar> limits = 15;
}
```

If the value of the limit for either CPU or memory is the IEEE “infinite” double value, then the task will be permitted to consume as much CPU and/or memory as is available on the machine - this is known as “unconstrained bursting”.

We use a ‘map’ field instead of ‘repeated Resource’ because the ‘Resource’ message contains a lot of metadata that we don’t need in order to define resource limits. We make use of the same type of map in the quota API for quota limits.

A new task status reason will be added for the case where a task is OOM-killed while exceeding its memory request:

```
message TaskStatus
{
  . . .
```

```
  enum Reason {
    . . .

    REASON_MEMORY_REQUEST_EXCEEDED = 35;
  }
}
```

The Mesos agent LAUNCH_CONTAINER call will also be updated with a ‘limits’ field:

```
message LaunchContainer {
  . . .

  map<string, Value.Scalar> limits = 5;
}
```

Isolator Interface

The ‘update()’ method of the Mesos containerizer’s isolator interface will be updated to accommodate resource limits:

```
  virtual process::Future<Nothing> update(
      const ContainerID& containerId,
      const Resources& resources,
      const Option<std::map<std::string, Value.Scalar>>& limits = None());
```

Nested cgroups

The proposal in this document is coupled to another feature: the addition of nested cgroup support in the CPU and memory cgroup subsystem isolators. As part of this work, a new parameter will be added to the ‘LinuxInfo’ message:

```
message LinuxInfo {
  . . .

  optional bool share_cgroups = 8 [default = true];
}
```

The ‘share_cgroups’ field should only be set for task groups; in other words, this field should only be set within subfields of the ‘LaunchGroup’ message. This will be validated at task/task group submission time, so that any ‘TaskInfo’ for a command task or Docker task which contains

a ‘LinuxInfo’ which has this field set will be invalidated, and a TASK_ERROR status update will be sent in response. If a framework does not set the ‘ExecutorInfo’ within ‘LaunchGroup.task_group.tasks’, the agent will copy the ‘ExecutorInfo’ into the tasks before sending them to the executor so that the executor will know whether or not it should launch them with nested cgroups.

Furthermore, a given executor may only use a single type of cgroup isolation. If a framework attempts to submit the same executor ID with a different type of isolation in the future, it will be invalidated.

The tables below summarize the cgroup behavior which will be observed by task groups in various configurations:

	Task resource limit set	Task resource limit not set
share_cgroups == false	Nested container gets its own cgroups, soft limits set to the resource requests and hard limits set to the resource limits. Task resource requests and limits will be counted into the executor cgroup’s soft and hard limits respectively.	Nested container gets its own cgroups, both soft and hard limits set to the resource requests. Task resource requests will be counted into the executor cgroup’s soft limits and also hard limits.
share_cgroups == true	TASK_ERROR	Old behavior: task shares executor cgroups, task resource requests will be counted into the executor cgroup’s soft limits and also hard limits.

The tables below summarize the cgroup behavior which will be observed by tasks (not in a task group) in various configurations:

	Task resource limit set	Task resource limit not set
share_cgroups is not set (it is true by default)	Task shares executor cgroups, executor cgroup’s soft and hard limits will be set to task resource requests and limits respectively.	Old behavior: task shares executor cgroups, both the executor cgroup’s soft limits and hard limits will be set to task resource requests.
share_cgroups is set	TASK_ERROR	TASK_ERROR

Implementation: MVP

Common Code

Stout's templated function 'protobuf::parse(const JSON::Value&)' will be updated to handle the case where a JSON string value is being parsed to a Protobuf double. If the string is equal to "Infinity" or "-Infinity", then the appropriate IEEE-defined double values will be used.

Similarly, stout's 'NumberWriter' in 'jsonify.hpp' will be updated to inspect the values of doubles when writing, and write the strings "Infinity" or "-Infinity" when the double has the value of positive or negative infinity.

Mesos Master

The master's validation code will be updated to:

- check that 'TaskInfo' only includes resource limits when the relevant agent possesses the TASK_RESOURCE_LIMITS capability.
- check that 'TaskInfo's within a 'TaskGroupInfo' only include resource limits when the task group's executor is the DEFAULT executor, using TASK isolation (or a custom executor).
- check the 'share_cgroups' field in 'LinuxInfo' when checking for equality of 'LinuxInfo' messages.
- check that the 'share_cgroups' field within 'LinuxInfo' is only set to 'false' within sub-messages that appear within 'LaunchGroup'.

Mesos Agent

The agent will be given a new capability, TASK_RESOURCE_LIMITS. This capability will be required for agent startup, and will be set by default.

The 'ContainerConfig' message will be extended to include the resource limits, if any, associated with the container:

```
message ContainerConfig
{
  . . .

  optional map<string, Value.Scalar> limits = 16;
}
```

The agent's LAUNCH_CONTAINER call chain will be updated to pass task limits through to the containerizer when launching containers. This call chain will also be updated to enforce that a

twice-nested container (i.e. a container whose parent also has a parent) cannot specify resource requests or limits. The only use cases for twice-nested containers are health checks and nested container sessions, and in both of these cases it makes sense for the

Mesos Isolator Interface

The signature of the isolator 'update()' method will be updated to include resource limits:

```
virtual process::Future<Nothing> update(
    const ContainerID& containerId,
    const ContainerUpdateParameters& parameters,
    const Option<std::map<std::string, Value.Scalar>>& limits = None());
```

Existing isolators will also be updated to reflect this change of signature.

Cgroups isolator

The cgroups isolator's 'prepare()' method will be updated to create nested cgroups for child containers when 'share_cgroups' is set to true in 'ContainerConfig'.

Linux CPU Subsystem Isolator

The CPU isolator's 'update()' method will be updated to set the container cgroup's 'cpu.shares' parameter according to the value of the container's CPU request (the sum of all CPU resources in 'ContainerConfig.resources'). It will set the container cgroup's 'cpu.cfs_quota_us' parameter according to the value of the container's CPU limit, if it exists. Note that the CFS quota will be set regardless of the value of the agent's '--cgroups_enable_cfs' flag. If the container's CPU limit is not set, then CFS quota will be set according to the old behavior, where it depends on the value of the agent flag.

Linux Memory Subsystem Isolator

The memory isolator's update()' method will be updated to set the container cgroup's 'memory.soft_limit_in_bytes' parameter according to the value of the container's memory request. It will set the container cgroup's 'memory.limit_in_bytes' parameter according to the value of the container's memory limit if it exists, or the container's memory request if the limit does not exist.

When there is enough memory pressure on a node that the OOM-killer is invoked, we would like to ensure that task processes which are currently consuming more memory than their memory request are preferentially killed first. To accomplish this, we can set '/proc/<pid>/oom_score_adj' for each task process after it is forked. In the memory isolator's 'isolate()' method, we will set the 'oom_score_adj' of a task to the following value, borrowed from the Kubernetes implementation:

```
1000 - (1000 * memoryRequest) / memoryCapacity
```

This heuristic attempts to ensure that tasks which exceed their memory request have an 'oom_score_adj' value of 1000, putting them first in line to be OOM-killed.

Docker Containerizer

The Docker library's 'RunOptions' will be extended with a member to hold resource limits, with a corresponding argument to 'Docker::RunOptions::create()'. 'Docker::run()' will be updated to set the '--memory', '--memory-reservation', '--cpu-shares', and '--cpu-quota' flags accordingly.

The Docker executor will be extended to set the 'oom_score_adj' of task processes using the '--oom_score_adj' flag for the Docker CLI.

The containerizer's 'update()' method will be updated to set the CPU shares, CFS quota, and memory hard/soft limits of a container to the values of its requests and limits when the 'limits' field is non-empty. This will be accomplished with the corresponding flags of the Docker CLI.

Default Executor

The default executor will be updated to use the LAUNCH_CONTAINER call instead of the LAUNCH_NESTED_CONTAINER call when launching nested containers. This will allow the default executor to set task limits when launching its task containers.

Mesos UI

The Mesos UI will be updated to display the resource requests and limits of tasks in the executor detail view, where task resources are currently displayed.

Implementation: Stretch Goal

While the above method of setting 'oom_score_adj' for task processes should provide satisfactory results in most cases, it will not be perfect. The calculation of process "badness" in the kernel also takes into account things like a process's running time and a process's capabilities. So there are likely certain sets of task configurations which, in the presence of memory pressure on a node, will lead to the kernel OOM-killing a task which is *not* exceeding its memory request *before* a task which *is* exceeding its memory request.

In order to provide a better guarantee that memory-bursting tasks will be OOM-killed before tasks using less than their memory request, we could dynamically set the 'oom_score_adj' of task processes. One mechanism for this is the cgroup notification API, and fortunately, Mesos already has cgroup notification listeners via the 'Listener' class in 'src/linux/cgroups.cpp'.

As a stretch goal, if we have time while implementing we could update the memory isolator's 'isolate()' method to create cgroup event listeners which are triggered whenever the memory cgroup's usage crosses the value of the container's memory request. This will be done iff the memory request is set to a different value than the memory limit. These listeners will trigger a callback which inspects the container's memory usage and sets the container process's '/proc/<pid>/oom_score_adj' value to 1000 when it has exceeded its memory request, or to zero when it is below its memory request.

If we do this work, we should also remove code from the 'prepare()' method which sets the 'oom_score_adj' statically, if it's already been added.

Open Questions

- How can we make REASON_CONTAINER_LIMITATION_MEMORY reliable? There is currently a race between OOM notification and process reaping.

Upgrades

While a cluster is in a partially-upgraded state, it's possible that an upgraded scheduler could send a 'TaskInfo' which specifies resource limits to an upgraded master, while the agent on which the task should be launched has not yet been upgraded. In such a case, the master will fail the task launch and send a TASK_ERROR status update to the scheduler.

Future Work

- In the future, we could allow "best-effort" tasks which specify no resource request at all for CPU and/or memory. Such a task could be launched on any agent regardless of available resources, and the task would be free to make use of any unused CPU and/or memory available on the agent.
- In the future, we could allow tasks to be run without specifying an executor at all. In such cases, we would use the default executor and we would not constrain the resource consumption of the executor container at all.
    - Note that in order for us to still provide task group-level limits, this would require us to have separate cgroups for the task group and for the executor (does the executor need a cgroup at all in this case?).
- In the future, we could allow resource limits to be specified at the task group level. This would allow the specification of task groups like "constrain containers A and B to 4 CPUs, but allow container C to use as much as is available".
- In the future, we could add code similar to Kubernetes' "eviction manager" to proactively kill task processes to reclaim memory before the OOM killer is invoked.