k8s resource control with cgroup

Cgroup has controllers that used to control many resources:

- blkio subsystem
- cpuacct subsystem
- cpuset subsystem
- devices subsystem
- freezer subsystem
- memory subsystem
- net_cls subsystem
- hugetlb subsystem

Now k8s only control the resource of cpu and memory, that means k8s only use cpu and memory cgroup controller. At k8s1.8, hugetlb will be supported.

CPU request and CPU limit:

```
apiVersion: v1
kind: Pod
metadata:
  name: cpu-demo
spec:
  containers:
  - name: cpu-demo
    Image: nginx
    resources:
      limits:
        cpu: "1"
      requests:
        cpu: "0.5"
```

- The `spec.containers[].resources.requests.cpu` is converted to its core value, which is potentially fractional, and multiplied by 1024. The greater of this number or 2 is used as the value of the `--cpu-shares` flag in the `docker run` command.
- The `spec.containers[].resources.limits.cpu` is converted to its millicore value and multiplied by 100. The resulting value is the total amount of CPU time

that a container can use every 100ms. A container cannot use more than its share of CPU time during this interval.

A Container might or might not be allowed to exceed its CPU limit for extended periods of time. However, it will not be killed for excessive CPU usage.

Memory request and memory limit:

```
apiVersion: v1
kind: Pod
metadata:
  name: memory-demo
spec:
  containers:
  - name: memory-demo
    image: nginx
    resources:
      limits:
        memory: "200Mi"
      requests:
        memory: "100Mi"
```

The `spec.containers[].resources.limits.memory` is converted to an integer, and used as the value of the [`--memory`](https://docs.docker.com/engine/reference/run/#user-memory-constraints) flag in the `docker run` command.

If a Container exceeds its memory limit, it might be terminated. If it is restartable, the kubelet will restart it, as with any other type of runtime failure.

If a Container exceeds its memory request, it is likely that its Pod will be evicted whenever the node runs out of memory.

CPU and Memory Resource control parameters

K8s resource	Docker resource	Cgroup	Description

name	name	resource	
cpu.request	--cpu-shares	cpu.shares	contains an integer value that specifies a relative share of CPU time available to the tasks in a cgroup.
cpu.limits	--cpu-quota	cpu.cfs_quota_us	specifies the total amount of time in microseconds (µs, represented here as "*us*") for which all tasks in a cgroup can run during one period (as defined by cpu.cfs_period_us). As soon as tasks in a cgroup use up all the time specified by the quota, they are throttled for the remainder of the time specified by the period and not allowed to run until the next period.
memory.limits	--memory	memory.limit_in_bytes	sets the maximum amount of user memory (including file cache).

Kubernetes Qos pod with cgroup

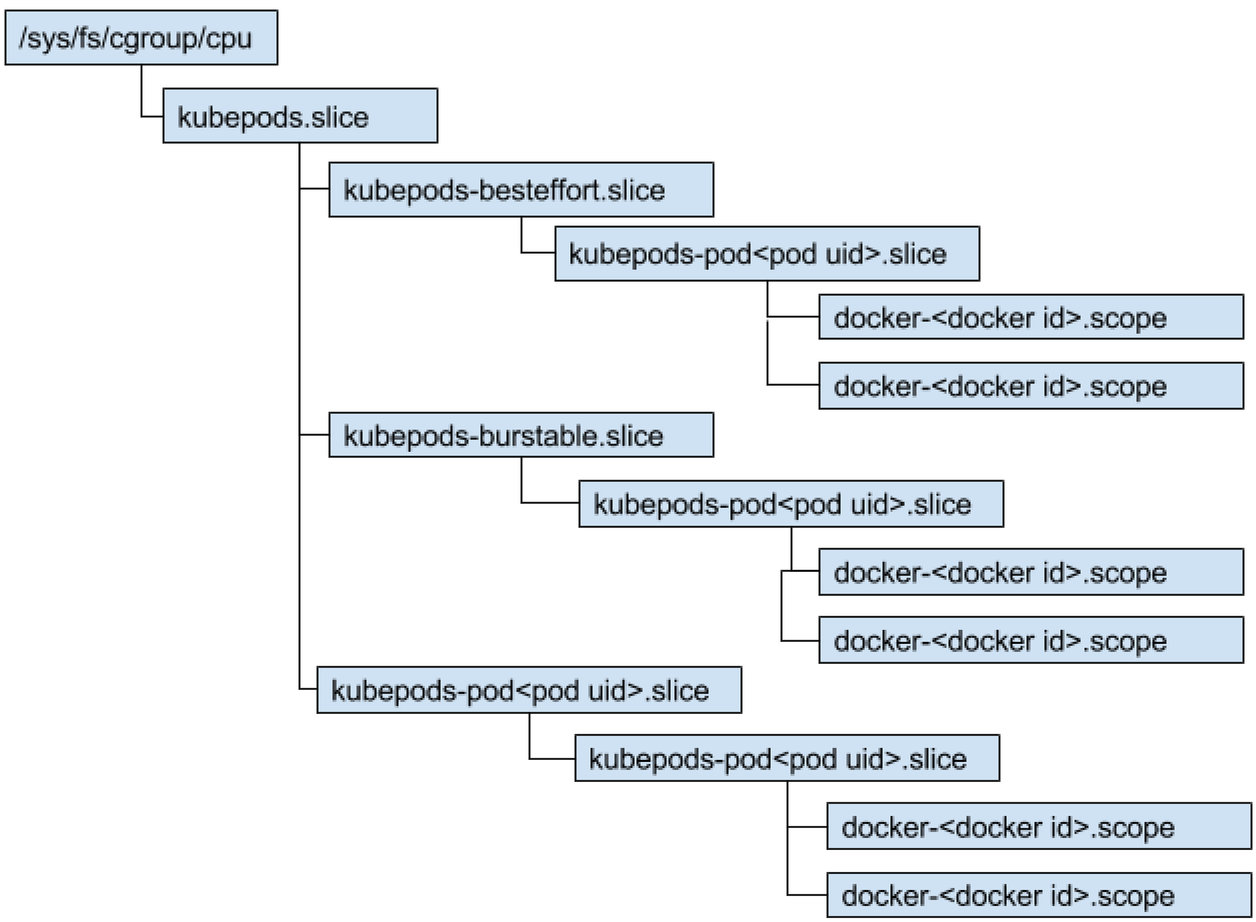
Based on the cpu , memory request and limit , the pods will be assigned particular Quality of Service (QoS) classes.

QoS level	Description
Best effort	The Containers in the Pod doesn't have any memory or cpu limits or requests.
Guaranteed	Every Container in the Pod must have a memory limit and a memory request, and they must be the same.

	Every Container in the Pod must have a cpu limit and a cpu request, and they must be the same.
Burstable	The Pod does not meet the criteria for QoS class Guaranteed. At least one Container in the Pod has a memory or cpu request.

Cgroup hierarchy in node for k8s 1.6

- **CPU hierarchy:**



/sys/fs/cgroup/cpu is the root cgroup. All of the pod’s cgroup is named Kubepods.slice as a child cgroup under root with machine.slice, user.slice and system.slice.

docker-<docker id>.scope config

cpu.shares	Best-effort	2
	Burstable	`spec.containers[].resources.requests.cpu * 1024`
	Guaranteed	`spec.containers[].resources.requests.cpu * 1024`
cpu.cfs_quota_us	Best-effort	`-1`
	Burstable	`Spec.containers[].resources.limits.cpu * 100`
	Guaranteed	`Spec.containers[].resources.limits.cpu * 100`

pod<pod id>.slice config

cpu.shares	Best-effort	2
	Burstable	Sum pod’s container (`spec.containers[].resources.requests.cpu * 1024)`
	Guaranteed	Sum pod’s container(`spec.containers[].resources.requests.cpu * 1024)`
cpu.cfs_quota_us	Best-effort	`-1`

	Burstable	SUM pod's container (Spec.containers[].resources.limits.cpu * 100)
	Guaranteed	SUM pod's container (Spec.containers[].resources.limits.cpu * 100)

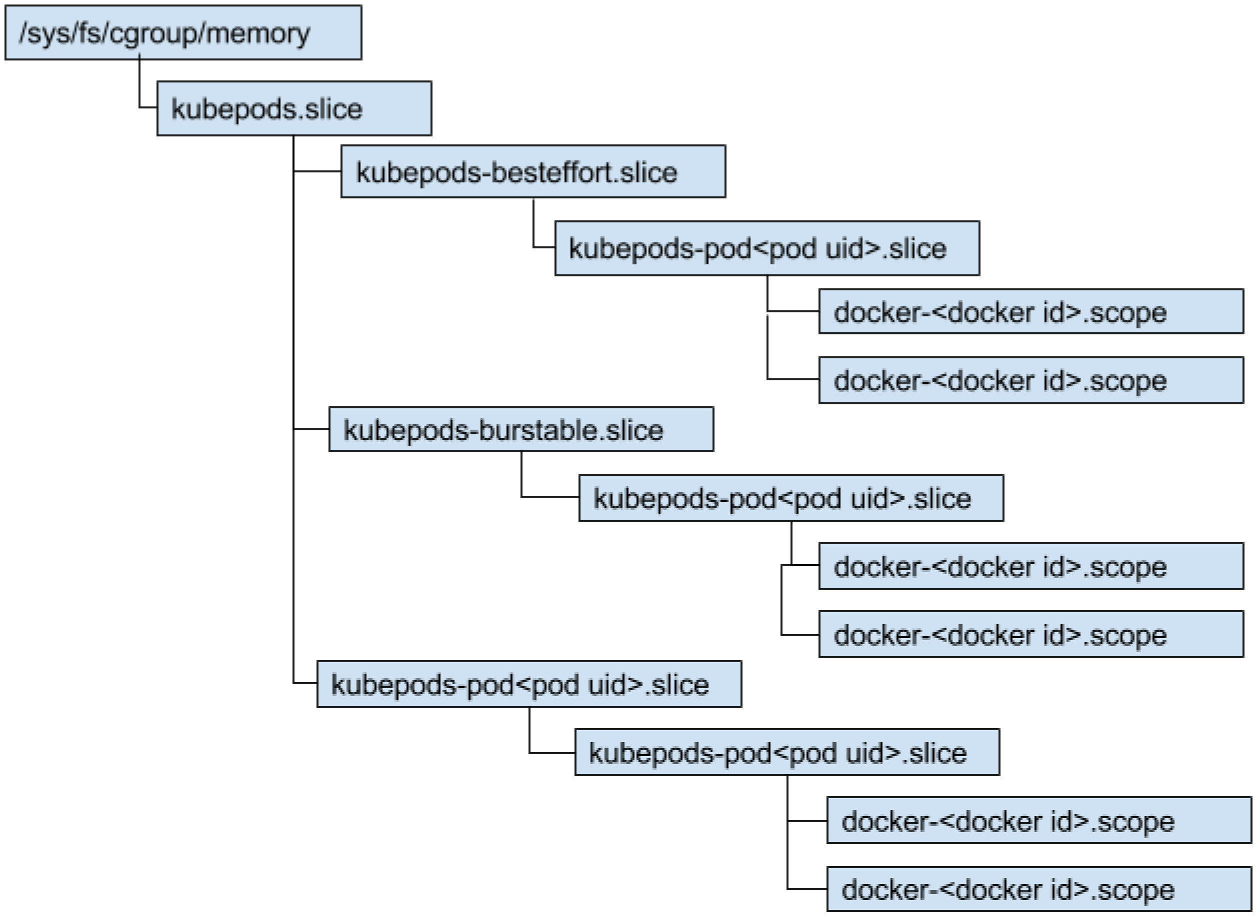
Kubepods-burstable.slice config

cpu.shares	SUM(spec.containers[].resources.requests.cpu * 1024) of all burstable pod
cpu.cfs_quota_us	-1

Kubepods-besteffort.slice config

cpu.shares	2
cpu.cfs_quota_us	-1

- **Memory hierarchy:**



The same hierarchy structure with cpu sub-system.

docker-<docker id>.scope config

memory.limit_in_bytes	Best-effort	Node memory
	Burstable	`spec.containers[].resources.limits.memory`
	Guaranteed	`spec.containers[].resources.limits.memory`

pod<pod id>.slice config

memory.limit_in_bytes	Best-effort	Node memory

	Burstable	Sum pod's container (`spec.containers[].resources.limits.memory`)
	Guaranteed	Sum pod's container(`spec.containers[].resources.limits.memory`)

Kubepods-burstable.slice config

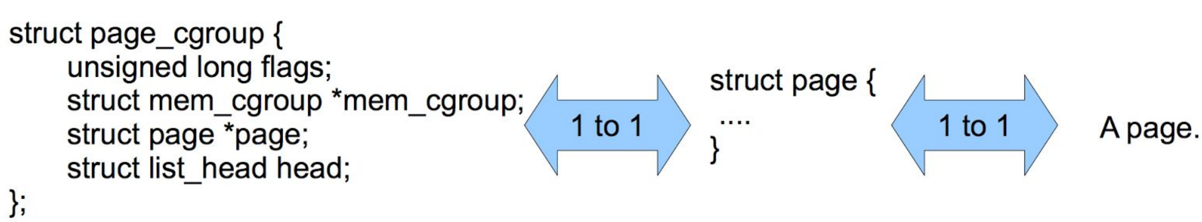
memory.limit_in_bytes	Node memory - Guaranteed memory

Kubepods-besteffort.slice config

memory.limit_in_bytes	Burstable memory - Burstable pod request Memory

Memory cgroup controller overhead for default pages:

- **Before 3.19**



Memcg uses page_cgroup for tracking all pages. It's allocated per page like struct page.

struct page_cgroup occupies 40bytes/4096bytes(x86-64), but usually we can't allocate all of the memory, so the maximum overhead is 1% of memory. This can be turned off by boot option: cgroup_disable=memory

- **After 3.19**

Each page have a pointer in struct page:

```
struct page {
...
   #ifdef CONFIG_MEMCG
          struct mem_cgroup *mem_cgroup;
   #endif
…
}
```

It points to the mem cgroup controller, so there is only 8 bytes overhead for each page.
8 bytes/4096 bytes = 0.195% of memory.

Hugetlb cgroup controller overhead for hugepage:

It will reuse the private field of page struct, so there is no more memory overhead.

```
static inline struct hugetlb_cgroup *hugetlb_cgroup_from_page(struct page *page)
{
        VM_BUG_ON_PAGE(!PageHuge(page), page);

        if (compound_order(page) < HUGETLB_CGROUP_MIN_ORDER)
                return NULL;
        return (struct hugetlb_cgroup *)page[2].private;
}

static inline
int set_hugetlb_cgroup(struct page *page, struct hugetlb_cgroup *h_cg)
{
        VM_BUG_ON_PAGE(!PageHuge(page), page);

        if (compound_order(page) < HUGETLB_CGROUP_MIN_ORDER)
                return -1;
        page[2].private         = (unsigned long)h_cg;
        return 0;
}
```

Memcg performance Test

Major performance impact of memcg is : Costs to modify system-wide shared counter for accounting usage

Memory charge	Page fault, file read, file write, swap-in, use a new page
Memory uncharge	Unmap, exit, truncate file, drop cache, kswapd… freeing a page

Test cases:

Cases	Operations
Kernel make on tmpfs	1. Make -j 64 2. No I/O layer influence, just for seeing cgroup cost
Page fault microbench	1. Do mmap/fault/unmap 256Mbytes of range 300 times. 2. Use anonymous and huge page memory, compare the result.

Test report:
https://docs.google.com/a/ebay.com/spreadsheets/d/10yssLXGdb6Smxwc5bRTaaJx7dxCKx0zi4XGYwkhY12U/edit?usp=sharing