Subject: Re: [PATCH v2 07/13] memcg: Slab accounting.
Posted by Glauber Costa on Sun, 11 Mar 2012 10:25:06 GMT
View Forum Message <> Reply to Message

On 03/10/2012 12:39 AM, Suleiman Souhlal wrote:
> Introduce per-cgroup kmem_caches for memcg slab accounting, that
> get created asynchronously the first time we do an allocation of
> that type in the cgroup.
> The cgroup cache gets used in subsequent allocations, and permits
> accounting of slab on a per-page basis.
>
> For a slab type to getaccounted, the SLAB_MEMCG_ACCT flag has to be
> passed to kmem_cache_create().
>
> The per-cgroup kmem_caches get looked up at slab allocation time,
> in a MAX_KMEM_CACHE_TYPES-sized array in the memcg structure, based
> on the original kmem_cache's id, which gets allocated when the original
> cache gets created.
>
> Only allocations that can be attributed to a cgroup get charged.
>
> Each cgroup kmem_cache has a refcount that dictates the lifetime
> of the cache: We destroy a cgroup cache when its cgroup has been
> destroyed and there are no more active objects in the cache.
>
> Signed-off-by: Suleiman Souhlal<suleiman@google.com>
> ---
> include/linux/memcontrol.h | 30 +++++-
> include/linux/slab.h | 41 ++++++
> include/linux/slab_def.h | 72 ++++++++++++-
> include/linux/slub_def.h | 3 +
> init/Kconfig | 2 +-
> mm/memcontrol.c | 290 ++
> mm/slab.c | 248 ++---
> 7 files changed, 663 insertions(+), 23 deletions(-)
>
> diff --git a/include/linux/memcontrol.h b/include/linux/memcontrol.h
> index b80de52..b6b8388 100644
> --- a/include/linux/memcontrol.h
> +++ b/include/linux/memcontrol.h
> @@ -416,13 +416,41 @@ struct sock;
> #ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
> void sock_update_memcg(struct sock *sk);
> void sock_release_memcg(struct sock *sk);
> -#else
> +struct kmem_cache *mem_cgroup_get_kmem_cache(struct kmem_cache *cachep,
> +   gfp_t gfp);
> +bool mem_cgroup_charge_slab(struct kmem_cache *cachep, gfp_t gfp, size_t size);

> +void mem_cgroup_uncharge_slab(struct kmem_cache *cachep, size_t size);
> +void mem_cgroup_flush_cache_create_queue(void);
> +void mem_cgroup_remove_child_kmem_cache(struct kmem_cache *cachep, int id);
> +#else /* CONFIG_CGROUP_MEM_RES_CTLR_KMEM */
> static inline void sock_update_memcg(struct sock *sk)
> {
> }
> static inline void sock_release_memcg(struct sock *sk)
> {
> }
> +
> +static inline bool
> +mem_cgroup_charge_slab(struct kmem_cache *cachep, gfp_t gfp, size_t size)
> +{
> + return true;
> +}
> +
> +static inline void
> +mem_cgroup_uncharge_slab(struct kmem_cache *cachep, size_t size)
> +{
> +}
> +
> +static inline struct kmem_cache *
> +mem_cgroup_get_kmem_cache(struct kmem_cache *cachep, gfp_t gfp)
> +{
> + return cachep;
> +}
> +
> +static inline void
> +mem_cgroup_flush_cache_create_queue(void)
> +{
> +}
> #endif /* CONFIG_CGROUP_MEM_RES_CTLR_KMEM */
> #endif /* _LINUX_MEMCONTROL_H */
>
> diff --git a/include/linux/slab.h b/include/linux/slab.h
> index 573c809..bc9f87f 100644
> --- a/include/linux/slab.h
> +++ b/include/linux/slab.h
> @@ -21,6 +21,7 @@
> #define SLAB_POISON 0x00000800UL /* DEBUG: Poison objects */
> #define SLAB_HWCACHE_ALIGN 0x00002000UL /* Align objs on cache lines */
> #define SLAB_CACHE_DMA 0x00004000UL /* Use GFP_DMA memory */
> +#define SLAB_MEMCG_ACCT 0x00008000UL /* Accounted by memcg */
> #define SLAB_STORE_USER 0x00010000UL /* DEBUG: Store the last owner for bug hunting
*/
> #define SLAB_PANIC 0x00040000UL /* Panic if kmem_cache_create() fails */
> /*

```
> @@ -153,6 +154,21 @@ unsigned int kmem_cache_size(struct kmem_cache *);
>   #define ARCH_SLAB_MINALIGN __alignof__(unsigned long long)
>   #endif
>
> +#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
> +struct mem_cgroup_cache_params {
> + struct mem_cgroup *memcg;
> + int id;
> + int obj_size;
> +
> + /* Original cache parameters, used when creating a memcg cache */
> + size_t orig_align;
> + unsigned long orig_flags;
> + struct kmem_cache *orig_cache;
> +
> + struct list_head destroyed_list; /* Used when deleting cpuset cache */
> +};
> +#endif
> +
>   /*
>    * Common kmalloc functions provided by all allocators
>    */
> @@ -353,4 +369,29 @@ static inline void *kzalloc_node(size_t size, gfp_t flags, int node)
>
>   void __init kmem_cache_init_late(void);
>
> +#if defined(CONFIG_CGROUP_MEM_RES_CTLR_KMEM)&&  defined(CONFIG_SLAB)
> +
> +#define MAX_KMEM_CACHE_TYPES 400
> +
> +struct kmem_cache *kmem_cache_create_memcg(struct kmem_cache *cachep,
> +    char *name);
> +void kmem_cache_drop_ref(struct kmem_cache *cachep);
> +
> +#else /* !CONFIG_CGROUP_MEM_RES_CTLR_KMEM || !CONFIG_SLAB */


> +#define MAX_KMEM_CACHE_TYPES 0
> +
> +static inline struct kmem_cache *
> +kmem_cache_create_memcg(struct kmem_cache *cachep,
> +    char *name)
> +{
> + return NULL;
> +}
> +
> +static inline void
> +kmem_cache_drop_ref(struct kmem_cache *cachep)
```

> +{
> +}
> +#endif /* CONFIG_CGROUP_MEM_RES_CTLR_KMEM&& CONFIG_SLAB */
> +
> #endif /* _LINUX_SLAB_H */
> diff --git a/include/linux/slab_def.h b/include/linux/slab_def.h
> index 25f9a6a..248b8a9 100644
> --- a/include/linux/slab_def.h
> +++ b/include/linux/slab_def.h
> @@ -51,7 +51,7 @@ struct kmem_cache {
> void (*ctor)(void *obj);
>
> /* 4) cache creation/removal */
> - const char *name;
> + char *name;
> struct list_head next;
>
> /* 5) statistics */
> @@ -81,6 +81,12 @@ struct kmem_cache {
> int obj_size;
> #endif /* CONFIG_DEBUG_SLAB */
>
> +#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
> + struct mem_cgroup_cache_params memcg_params;
> +
Can't we have memcg pointer + id here, and then get
the rest of this data from memcg->slab_params[id] ?

I don't think we'll ever access this on hot paths - since we're
allocating pages anyway when we do, and then we have less bloat on
the slab structure.

Sure, we're just moving the bloat away somewhere else, since this data
has to exist anyway, but at least it's not here bothering the slab when
memcg is disabled...

But I don't really have the final word here, this is just my preference
- Would any of the slabmasters comment here, on what's your preferred
way between those?

> + atomic_t refcnt;
> +#endif /* CONFIG_CGROUP_MEM_RES_CTLR_KMEM */
> +
> /* 6) per-cpu/per-node data, touched during every alloc/free */
> /*
> * We put array[] at the end of kmem_cache, because we want to size
> @@ -218,4 +224,68 @@ found:
>

> #endif /* CONFIG_NUMA */
>
> +#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
> +
> +void kmem_cache_drop_ref(struct kmem_cache *cachep);
> +
> +static inline void
> +kmem_cache_get_ref(struct kmem_cache *cachep)
> +{
> + if (cachep->memcg_params.id == -1&&
> +   unlikely(!atomic_add_unless(&cachep->refcnt, 1, 0)))
> + BUG();
> +}
> +
> +static inline void
> +mem_cgroup_put_kmem_cache(struct kmem_cache *cachep)
> +{
> + rcu_read_unlock();
> +}
> +
> +static inline void
> +mem_cgroup_kmem_cache_prepare_sleep(struct kmem_cache *cachep)
> +{
> + /*
> + * Make sure the cache doesn't get freed while we have interrupts
> + * enabled.
> + */
> + kmem_cache_get_ref(cachep);
> + rcu_read_unlock();
> +}

Is this really needed ? After this function call in slab.c, the slab
code itself accesses cachep a thousand times. If it could be freed, it
would already explode today for other reasons?
Am I missing something here?

> +static inline void
> +mem_cgroup_kmem_cache_finish_sleep(struct kmem_cache *cachep)
> +{
> + rcu_read_lock();
> + kmem_cache_drop_ref(cachep);
> +}
> +
> +#else /* CONFIG_CGROUP_MEM_RES_CTLR_KMEM */
> +
> +static inline void
> +kmem_cache_get_ref(struct kmem_cache *cachep)
> +{

> +}
> +
> +static inline void
> +kmem_cache_drop_ref(struct kmem_cache *cachep)
> +{
> +}
> +
> +static inline void
> +mem_cgroup_put_kmem_cache(struct kmem_cache *cachep)
> +{
> +}
> +
> +static inline void
> +mem_cgroup_kmem_cache_prepare_sleep(struct kmem_cache *cachep)
> +{
> +}
> +
> +static inline void
> +mem_cgroup_kmem_cache_finish_sleep(struct kmem_cache *cachep)
> +{
> +}
> +#endif /* CONFIG_CGROUP_MEM_RES_CTLR_KMEM */
> +
> #endif /* _LINUX_SLAB_DEF_H */
> diff --git a/include/linux/slub_def.h b/include/linux/slub_def.h
> index 5911d81..09b602d 100644
> --- a/include/linux/slub_def.h
> +++ b/include/linux/slub_def.h
> @@ -99,6 +99,9 @@ struct kmem_cache {
> #ifdef CONFIG_SYSFS
>   struct kobject kobj; /* For sysfs */
> #endif
> +#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
> + struct mem_cgroup_cache_params memcg_params;
> +#endif
>
> #ifdef CONFIG_NUMA
>   /*
> diff --git a/init/Kconfig b/init/Kconfig
> index 3f42cd6..e7eb652 100644
> --- a/init/Kconfig
> +++ b/init/Kconfig
> @@ -705,7 +705,7 @@ config CGROUP_MEM_RES_CTLR_SWAP_ENABLED
>   then swapaccount=0 does the trick).
> config CGROUP_MEM_RES_CTLR_KMEM
>   bool "Memory Resource Controller Kernel Memory accounting (EXPERIMENTAL)"
> - depends on CGROUP_MEM_RES_CTLR&& EXPERIMENTAL
> + depends on CGROUP_MEM_RES_CTLR&& EXPERIMENTAL&& !SLOB

Orthogonal question: Will we ever want this (SLOB) ?

> default n
> help
> The Kernel Memory extension for Memory Resource Controller can limit
> diff --git a/mm/memcontrol.c b/mm/memcontrol.c
> index 2576a2b..a5593cf 100644
> --- a/mm/memcontrol.c
> +++ b/mm/memcontrol.c
> @@ -302,6 +302,11 @@ struct mem_cgroup {
> #ifdef CONFIG_INET
> struct tcp_memcontrol tcp_mem;
> #endif
> +
> +#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
> + /* Slab accounting */
> + struct kmem_cache *slabs[MAX_KMEM_CACHE_TYPES];
> +#endif
> };
>
> /* Stuffs for move charges at task migration. */
> @@ -5692,6 +5697,287 @@ memcg_uncharge_kmem(struct mem_cgroup *memcg, long long delta)
> res_counter_uncharge(&memcg->res, delta);
> }
>
> +static struct kmem_cache *
> +memcg_create_kmem_cache(struct mem_cgroup *memcg, struct kmem_cache *cachep)
> +{
> + struct kmem_cache *new_cachep;
> + struct dentry *dentry;
> + char *name;
> + int idx;
> +
> + idx = cachep->memcg_params.id;
> +
> + dentry = memcg->css.cgroup->dentry;
> + BUG_ON(dentry == NULL);
> +
> + /* Preallocate the space for "dead" at the end */
> + name = kasprintf(GFP_KERNEL, "%s(%d:%s)dead",
> + cachep->name, css_id(&memcg->css), dentry->d_name.name);
> + if (name == NULL)
> + return cachep;
> + /* Remove "dead" */
> + name[strlen(name) - 4] = '\0';
> +

> + new_cachep = kmem_cache_create_memcg(cachep, name);
> +
> + /*
> + * Another CPU is creating the same cache?
> + * We'll use it next time.
> + */
This comment is a bit misleading. Is it really the only reason
it can fail?

The impression I got is that it can also fail under the normal
conditions in which kmem_cache_create() fails.

> + if (new_cachep == NULL) {
> + kfree(name);
> + return cachep;
> + }
> +
> + new_cachep->memcg_params.memcg = memcg;
> +
> + /*
> + * Make sure someone else hasn't created the new cache in the
> + * meantime.
> + * This should behave as a write barrier, so we should be fine
> + * with RCU.
> + */
> + if (cmpxchg(&memcg->slabs[idx], NULL, new_cachep) != NULL) {
> + kmem_cache_destroy(new_cachep);
> + return cachep;
> + }
> +
> + return new_cachep;
> +}
> +
> +struct create_work {
> + struct mem_cgroup *memcg;
> + struct kmem_cache *cachep;
> + struct list_head list;
> +};
> +
> +static DEFINE_SPINLOCK(create_queue_lock);
> +static LIST_HEAD(create_queue);

If we move memcg_params to somewhere inside memcontrol.c, we can put
this worker inside that struct, and save the mallocs here.

We can have it separated from the main mem_cgroup, like the thresholds.
Another way is to have a zero-sized array living at the end of struct
mem_cgroup. Then we allocate sizeof(*memcg) for the root mem cgroup,

and sizeof(*memcg) + sizeof(max_cache_size) for the rest.

```
> +/*
> + * Flush the queue of kmem_caches to create, because we're creating a cgroup.
> + *
> + * We might end up flushing other cgroups' creation requests as well, but
> + * they will just get queued again next time someone tries to make a slab
> + * allocation for them.
> + */
> +void
> +mem_cgroup_flush_cache_create_queue(void)
> +{
> + struct create_work *cw, *tmp;
> + unsigned long flags;
> +
> + spin_lock_irqsave(&create_queue_lock, flags);
> + list_for_each_entry_safe(cw, tmp,&create_queue, list) {
> +  list_del(&cw->list);
> +  kfree(cw);
> + }
> + spin_unlock_irqrestore(&create_queue_lock, flags);
> +}
> +
> +static void
> +memcg_create_cache_work_func(struct work_struct *w)
> +{
> + struct kmem_cache *cachep;
> + struct create_work *cw;
> +
> + spin_lock_irq(&create_queue_lock);
> + while (!list_empty(&create_queue)) {
> +  cw = list_first_entry(&create_queue, struct create_work, list);
> +  list_del(&cw->list);
> +  spin_unlock_irq(&create_queue_lock);
> +  cachep = memcg_create_kmem_cache(cw->memcg, cw->cachep);
> +  if (cachep == NULL&&  printk_ratelimit())
> +   printk(KERN_ALERT "%s: Couldn't create memcg-cache for"
> +       " %s memcg %s\n", __func__, cw->cachep->name,
> +       cw->memcg->css.cgroup->dentry->d_name.name);
> +  /* Drop the reference gotten when we enqueued. */
> +  css_put(&cw->memcg->css);
> +  kfree(cw);
> +  spin_lock_irq(&create_queue_lock);
> + }
> + spin_unlock_irq(&create_queue_lock);
> +}
> +
> +static DECLARE_WORK(memcg_create_cache_work, memcg_create_cache_work_func);
```

> +
> +/*
> + * Enqueue the creation of a per-memcg kmem_cache.
> + * Called with rcu_read_lock.
> + */
> +static void
> +memcg_create_cache_enqueue(struct mem_cgroup *memcg, struct kmem_cache *cachep)
> +{
> + struct create_work *cw;
> + unsigned long flags;
> +
> + spin_lock_irqsave(&create_queue_lock, flags);
If we can sleep, why not just create the cache now?

Maybe it would be better to split this in two, and create the cache if
possible, and a worker if not possible. Then w

> + list_for_each_entry(cw,&create_queue, list) {
> + if (cw->memcg == memcg&& cw->cachep == cachep) {
> +   spin_unlock_irqrestore(&create_queue_lock, flags);
> +   return;
> + }
> + }
> + spin_unlock_irqrestore(&create_queue_lock, flags);
> +
> + /* The corresponding put will be done in the workqueue. */
> + if (!css_tryget(&memcg->css))
> + return;
> + cw = kmalloc(sizeof(struct create_work), GFP_NOWAIT);
> + if (cw == NULL) {
> + css_put(&memcg->css);
> + return;
> + }
> +
> + cw->memcg = memcg;
> + cw->cachep = cachep;
> + spin_lock_irqsave(&create_queue_lock, flags);
> + list_add_tail(&cw->list,&create_queue);
> + spin_unlock_irqrestore(&create_queue_lock, flags);
> +
> + schedule_work(&memcg_create_cache_work);
> +}
> +
> +/*
> + * Return the kmem_cache we're supposed to use for a slab allocation.
> + * If we are in interrupt context or otherwise have an allocation that
> + * can't fail, we return the original cache.
> + * Otherwise, we will try to use the current memcg's version of the cache.

> + *
> + * If the cache does not exist yet, if we are the first user of it,
> + * we either create it immediately, if possible, or create it asynchronously
> + * in a workqueue.
> + * In the latter case, we will let the current allocation go through with
> + * the original cache.
> + *
> + * This function returns with rcu_read_lock() held.
> + */
> +struct kmem_cache *
> +mem_cgroup_get_kmem_cache(struct kmem_cache *cachep, gfp_t gfp)
> +{
> + struct mem_cgroup *memcg;
> + int idx;
> +
> + rcu_read_lock();
> +
> + if (in_interrupt())
> + return cachep;
> + if (current == NULL)
> + return cachep;
> + if (!(cachep->flags& SLAB_MEMCG_ACCT))
> + return cachep;
> +
> + gfp |= kmem_cache_gfp_flags(cachep);
> + if (gfp& __GFP_NOFAIL)
> + return cachep;
> +
> + if (cachep->memcg_params.memcg)
> + return cachep;
> +
> + memcg = mem_cgroup_from_task(current);
> + idx = cachep->memcg_params.id;
> +
> + if (memcg == NULL || memcg == root_mem_cgroup)
nitpick: mem_cgroup_is_root(memcg)

It would also be better to have this test done as early as possible,
to reduce the penalty for the root memcg case.

We also need to test mem_cgroup_disabled() and return cachep in
this case.

> + return cachep;
> +
> + VM_BUG_ON(idx == -1);
> +
> + if (rcu_access_pointer(memcg->slabs[idx]) == NULL) {

```
if (unlikely(rcu_access_pointer(...

> +  memcg_create_cache_enqueue(memcg, cachep);
> +  return cachep;
> + }
> +
> + return rcu_dereference(memcg->slabs[idx]);
> +}
> +
> +void
> +mem_cgroup_remove_child_kmem_cache(struct kmem_cache *cachep, int id)
> +{
> + rcu_assign_pointer(cachep->memcg_params.memcg->slabs[id], NULL);
> +}
> +
> +bool
> +mem_cgroup_charge_slab(struct kmem_cache *cachep, gfp_t gfp, size_t size)
> +{
> + struct mem_cgroup *memcg;
> + int ret;
> +
> + rcu_read_lock();
> + memcg = cachep->memcg_params.memcg;

        if (!memcg) {
            rcu_read_unlock();
            return 0;
        }

> + if (memcg&&  !css_tryget(&memcg->css))
> +  memcg = NULL;
> + rcu_read_unlock();
> +
        if (!memcg)
            return...

> + ret = memcg_charge_kmem(memcg, gfp, size);
> + if (memcg)
> +  css_put(&memcg->css);
> +
> + return ret == 0;
> +}
> +
> +void
> +mem_cgroup_uncharge_slab(struct kmem_cache *cachep, size_t size)
> +{
> + struct mem_cgroup *memcg;
> +
```

```
> + rcu_read_lock();
> + memcg = cachep->memcg_params.memcg;
> +
> + if (memcg&&  !css_tryget(&memcg->css))
> +  memcg = NULL;
> + rcu_read_unlock();
> +
> + memcg_uncharge_kmem(memcg, size);
> + if (memcg)
> +  css_put(&memcg->css);
> +}
> +
> +static void
> +memcg_slab_init(struct mem_cgroup *memcg)
> +{
> + int i;
> +
> + for (i = 0; i<  MAX_KMEM_CACHE_TYPES; i++)
> +  rcu_assign_pointer(memcg->slabs[i], NULL);
> +}
> +
> +/*
> + * Mark all of this memcg's kmem_caches as dead and move them to the
> + * root.
> + *
> + * Assumes that the callers are synchronized (only one thread should be
> + * moving a cgroup's slab at the same time).
> + */
> +static void
> +memcg_slab_move(struct mem_cgroup *memcg)
> +{
> + struct kmem_cache *cachep;
> + int i;
> +
> + mem_cgroup_flush_cache_create_queue();
> +
> + for (i = 0; i<  MAX_KMEM_CACHE_TYPES; i++) {
> +  cachep = rcu_access_pointer(memcg->slabs[i]);
> +  if (cachep != NULL) {
> +   rcu_assign_pointer(memcg->slabs[i], NULL);
> +   cachep->memcg_params.memcg = NULL;
> +
> +   /* The space for this is already allocated */
> +   strcat((char *)cachep->name, "dead");
> +
> +   /*
> +    * Drop the initial reference on the cache.
> +    * This means that from this point on, the cache will
```

```
> +    * get destroyed when it no longer has active objects.
> +    */
> +   kmem_cache_drop_ref(cachep);
> +  }
> + }
> +}
> +
>   static void
>   memcg_kmem_init(struct mem_cgroup *memcg, struct mem_cgroup *parent)
>   {
> @@ -5701,6 +5987,8 @@ memcg_kmem_init(struct mem_cgroup *memcg, struct mem_cgroup
*parent)
>    if (parent&&  parent != root_mem_cgroup)
>     parent_res =&parent->kmem;
>    res_counter_init(&memcg->kmem, parent_res);
> +
> + memcg_slab_init(memcg);
>   }
>
>   static void
> @@ -5709,6 +5997,8 @@ memcg_kmem_move(struct mem_cgroup *memcg)
>    unsigned long flags;
>    long kmem;
>
> + memcg_slab_move(memcg);
> +
>    spin_lock_irqsave(&memcg->kmem.lock, flags);
>    kmem = memcg->kmem.usage;
>    res_counter_uncharge_locked(&memcg->kmem, kmem);
> diff --git a/mm/slab.c b/mm/slab.c
> index f0bd785..95b024c 100644
> --- a/mm/slab.c
> +++ b/mm/slab.c
> @@ -159,13 +159,15 @@
>       SLAB_STORE_USER | \
>       SLAB_RECLAIM_ACCOUNT | SLAB_PANIC | \
>       SLAB_DESTROY_BY_RCU | SLAB_MEM_SPREAD | \
> -    SLAB_DEBUG_OBJECTS | SLAB_NOLEAKTRACE | SLAB_NOTRACK)
> +    SLAB_DEBUG_OBJECTS | SLAB_NOLEAKTRACE | \
> +    SLAB_NOTRACK | SLAB_MEMCG_ACCT)
>   #else
>   # define CREATE_MASK (SLAB_HWCACHE_ALIGN | \
>       SLAB_CACHE_DMA | \
>       SLAB_RECLAIM_ACCOUNT | SLAB_PANIC | \
>       SLAB_DESTROY_BY_RCU | SLAB_MEM_SPREAD | \
> -    SLAB_DEBUG_OBJECTS | SLAB_NOLEAKTRACE | SLAB_NOTRACK)
> +    SLAB_DEBUG_OBJECTS | SLAB_NOLEAKTRACE | \
> +    SLAB_NOTRACK | SLAB_MEMCG_ACCT)
```

> #endif
>
> /*
> @@ -301,6 +303,8 @@ static void free_block(struct kmem_cache *cachep, void **objpp, int
len,
>   int node);
> static int enable_cpucache(struct kmem_cache *cachep, gfp_t gfp);
> static void cache_reap(struct work_struct *unused);
> +static int do_tune_cpucache(struct kmem_cache *cachep, int limit,
> +   int batchcount, int shared, gfp_t gfp);
>
> /*
>   * This function must be completely optimized away if a constant is passed to
> @@ -326,6 +330,11 @@ static __always_inline int index_of(const size_t size)
>   return 0;
> }
>
> +#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
> +/* Bitmap used for allocating the cache id numbers. */
> +static DECLARE_BITMAP(cache_types, MAX_KMEM_CACHE_TYPES);
> +#endif
> +
> static int slab_early_init = 1;
>
> #define INDEX_AC index_of(sizeof(struct arraycache_init))
> @@ -1756,17 +1765,23 @@ static void *kmem_getpages(struct kmem_cache *cachep, gfp_t
flags, int nodeid)
>   if (cachep->flags& SLAB_RECLAIM_ACCOUNT)
>   flags |= __GFP_RECLAIMABLE;
>
> + nr_pages = (1<< cachep->gfporder);
> + if (!mem_cgroup_charge_slab(cachep, flags, nr_pages * PAGE_SIZE))
> + return NULL;
> +
>   page = alloc_pages_exact_node(nodeid, flags | __GFP_NOTRACK, cachep->gfporder);
> - if (!page)
> + if (!page) {
> + mem_cgroup_uncharge_slab(cachep, nr_pages * PAGE_SIZE);
>   return NULL;
> + }

Can't the following happen:

*) mem_cgroup_charge_slab() is the first one to touch the slab.
   Therefore, this first one is billed to root.
*) A slab is queued for creation.
*) alloc_pages sleep.

*) our workers run, and create the cache, therefore filling
   cachep->memcg_param.memcg
*) alloc_pages still can't allocate.
*) uncharge tries to uncharge from cachep->memcg_param.memcg,
   which doesn't have any charges...

Unless you have a strong oposition to this, to avoid this kind of
corner cases, we could do what I was doing in the slub:
Allocate the page first, and then account it.
(freeing the page if it fails).

I know it is not the way it is done for the user pages, but I believe it
to be better suited for the slab.

> - nr_pages = (1<< cachep->gfporder);
>   if (cachep->flags& SLAB_RECLAIM_ACCOUNT)
>   add_zone_page_state(page_zone(page),
>   NR_SLAB_RECLAIMABLE, nr_pages);
>   else
>   add_zone_page_state(page_zone(page),
>   NR_SLAB_UNRECLAIMABLE, nr_pages);
> + kmem_cache_get_ref(cachep);
>   for (i = 0; i< nr_pages; i++)
>   __SetPageSlab(page + i);
>
> @@ -1799,6 +1814,8 @@ static void kmem_freepages(struct kmem_cache *cachep, void
*addr)
>   else
>   sub_zone_page_state(page_zone(page),
>   NR_SLAB_UNRECLAIMABLE, nr_freed);
> + mem_cgroup_uncharge_slab(cachep, i * PAGE_SIZE);
> + kmem_cache_drop_ref(cachep);
>   while (i--) {
>   BUG_ON(!PageSlab(page));
>   __ClearPageSlab(page);
> @@ -2224,14 +2241,17 @@ static int __init_refok setup_cpu_cache(struct kmem_cache
*cachep, gfp_t gfp)
>   * cacheline. This can be beneficial if you're counting cycles as closely
>   * as davem.
>   */
> -struct kmem_cache *
> -kmem_cache_create (const char *name, size_t size, size_t align,
> - unsigned long flags, void (*ctor)(void *))
> +static struct kmem_cache *
> +__kmem_cache_create(const char *name, size_t size, size_t align,
> +   unsigned long flags, void (*ctor)(void *), bool memcg)
>   {
> - size_t left_over, slab_size, ralign;

> + size_t left_over, orig_align, ralign, slab_size;
>   struct kmem_cache *cachep = NULL, *pc;
> + unsigned long orig_flags;
>   gfp_t gfp;
>
> + orig_align = align;
> + orig_flags = flags;
>   /*
>   * Sanity checks... these are all serious usage bugs.
>   */
> @@ -2248,7 +2268,6 @@ kmem_cache_create (const char *name, size_t size, size_t align,
>   */
>   if (slab_is_available()) {
>   get_online_cpus();
> - mutex_lock(&cache_chain_mutex);
>   }
>
>   list_for_each_entry(pc,&cache_chain, next) {
> @@ -2269,10 +2288,12 @@ kmem_cache_create (const char *name, size_t size, size_t align,
>   }
>
>   if (!strcmp(pc->name, name)) {
> -   printk(KERN_ERR
> -   "kmem_cache_create: duplicate cache %s\n", name);
> -   dump_stack();
> -   goto oops;
> +   if (!memcg) {
> +   printk(KERN_ERR "kmem_cache_create: duplicate"
> +   " cache %s\n", name);
> +   dump_stack();
> +   goto oops;
> +   }

Why? Since we are apending the memcg name at the end anyway, duplicates
still aren't expected.

>   }
>   }
>
> @@ -2369,6 +2390,13 @@ kmem_cache_create (const char *name, size_t size, size_t align,
>   goto oops;
>
>   cachep->nodelists = (struct kmem_list3 **)&cachep->array[nr_cpu_ids];
> +
> +#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
> + cachep->memcg_params.obj_size = size;
> + cachep->memcg_params.orig_align = orig_align;
> + cachep->memcg_params.orig_flags = orig_flags;
> +#endif

```
> +
>   #if DEBUG
>    cachep->obj_size = size;
>
> @@ -2477,7 +2505,23 @@ kmem_cache_create (const char *name, size_t size, size_t align,
>     BUG_ON(ZERO_OR_NULL_PTR(cachep->slabp_cache));
>    }
>    cachep->ctor = ctor;
> - cachep->name = name;
> + cachep->name = (char *)name;
> +
> +#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
> + cachep->memcg_params.orig_cache = NULL;
> + atomic_set(&cachep->refcnt, 1);
> + INIT_LIST_HEAD(&cachep->memcg_params.destroyed_list);
> +
> + if (!memcg) {
> +  int id;
> +
> +  id = find_first_zero_bit(cache_types, MAX_KMEM_CACHE_TYPES);
> +  BUG_ON(id<  0 || id>= MAX_KMEM_CACHE_TYPES);
> +  __set_bit(id, cache_types);
> +  cachep->memcg_params.id = id;
> + } else
> +  cachep->memcg_params.id = -1;
> +#endif
>
>    if (setup_cpu_cache(cachep, gfp)) {
>     __kmem_cache_destroy(cachep);
> @@ -2502,13 +2546,53 @@ oops:
>     panic("kmem_cache_create(): failed to create slab `%s'\n",
>           name);
>    if (slab_is_available()) {
> -  mutex_unlock(&cache_chain_mutex);
>     put_online_cpus();
>    }
>    return cachep;
>   }
> +
> +struct kmem_cache *
> +kmem_cache_create(const char *name, size_t size, size_t align,
> +    unsigned long flags, void (*ctor)(void *))
> +{
> + struct kmem_cache *cachep;
> +
> + mutex_lock(&cache_chain_mutex);
> + cachep = __kmem_cache_create(name, size, align, flags, ctor, false);
> + mutex_unlock(&cache_chain_mutex);
```

```
> +
> + return cachep;
> +}
>   EXPORT_SYMBOL(kmem_cache_create);
>
> +#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
> +struct kmem_cache *
> +kmem_cache_create_memcg(struct kmem_cache *cachep, char *name)
> +{
> + struct kmem_cache *new;
> + int flags;
> +
> + flags = cachep->memcg_params.orig_flags&  ~SLAB_PANIC;
> + mutex_lock(&cache_chain_mutex);
> + new = __kmem_cache_create(name, cachep->memcg_params.obj_size,
> +     cachep->memcg_params.orig_align, flags, cachep->ctor, 1);
> + if (new == NULL) {
> +  mutex_unlock(&cache_chain_mutex);
> +  return NULL;
> + }
> + new->memcg_params.orig_cache = cachep;
> +
> + if ((cachep->limit != new->limit) ||
> +     (cachep->batchcount != new->batchcount) ||
> +     (cachep->shared != new->shared))
> +  do_tune_cpucache(new, cachep->limit, cachep->batchcount,
> +      cachep->shared, GFP_KERNEL);
> + mutex_unlock(&cache_chain_mutex);
> +
> + return new;
> +}
> +#endif /* CONFIG_CGROUP_MEM_RES_CTLR_KMEM */
> +
>   #if DEBUG
>   static void check_irq_off(void)
>   {
> @@ -2703,12 +2787,74 @@ void kmem_cache_destroy(struct kmem_cache *cachep)
>    if (unlikely(cachep->flags&  SLAB_DESTROY_BY_RCU))
>     rcu_barrier();
>
> +#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
> + /* Not a memcg cache */
> + if (cachep->memcg_params.id != -1) {
> +  __clear_bit(cachep->memcg_params.id, cache_types);
> +  mem_cgroup_flush_cache_create_queue();
> + }
> +#endif
```

This will clear the id when a leaf cache is destroyed. It seems it is
not what we want, right? We want this id to be cleared only when
the parent cache is gone.

```
>    __kmem_cache_destroy(cachep);
>    mutex_unlock(&cache_chain_mutex);
>    put_online_cpus();
>   }
>   EXPORT_SYMBOL(kmem_cache_destroy);
>
> +#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
> +static DEFINE_SPINLOCK(destroy_lock);
> +static LIST_HEAD(destroyed_caches);
> +
> +static void
> +kmem_cache_destroy_work_func(struct work_struct *w)
> +{
> + struct kmem_cache *cachep;
> + char *name;
> +
> + spin_lock_irq(&destroy_lock);
> + while (!list_empty(&destroyed_caches)) {
> +  cachep = container_of(list_first_entry(&destroyed_caches,
> +      struct mem_cgroup_cache_params, destroyed_list), struct
> +      kmem_cache, memcg_params);
> +  name = (char *)cachep->name;
> +  list_del(&cachep->memcg_params.destroyed_list);
> +  spin_unlock_irq(&destroy_lock);
> +  synchronize_rcu();
> +  kmem_cache_destroy(cachep);
                ^^^^^^
                will destroy the id.

> +  kfree(name);
> +  spin_lock_irq(&destroy_lock);
> + }
> + spin_unlock_irq(&destroy_lock);
> +}
> +
> +static DECLARE_WORK(kmem_cache_destroy_work, kmem_cache_destroy_work_func);
> +
> +static void
> +kmem_cache_destroy_memcg(struct kmem_cache *cachep)
> +{
> + unsigned long flags;
> +
> + BUG_ON(cachep->memcg_params.id != -1);
> +
```

```
> + /*
> +  * We have to defer the actual destroying to a workqueue, because
> +  * we might currently be in a context that cannot sleep.
> +  */
> + spin_lock_irqsave(&destroy_lock, flags);
> + list_add(&cachep->memcg_params.destroyed_list,&destroyed_caches);
> + spin_unlock_irqrestore(&destroy_lock, flags);
> +
> + schedule_work(&kmem_cache_destroy_work);
> +}
> +
> +void
> +kmem_cache_drop_ref(struct kmem_cache *cachep)
> +{
> + if (cachep->memcg_params.id == -1&&
> +     unlikely(atomic_dec_and_test(&cachep->refcnt)))
> +  kmem_cache_destroy_memcg(cachep);
> +}
> +#endif /* CONFIG_CGROUP_MEM_RES_CTLR_KMEM */
> +
>   /*
>    * Get the memory for a slab management obj.
>    * For a slab cache when the slab descriptor is off-slab, slab descriptors
> @@ -2908,8 +3054,10 @@ static int cache_grow(struct kmem_cache *cachep,
>
>    offset *= cachep->colour_off;
>
> - if (local_flags&  __GFP_WAIT)
> + if (local_flags&  __GFP_WAIT) {
>     local_irq_enable();
> +  mem_cgroup_kmem_cache_prepare_sleep(cachep);
> + }
>
>    /*
>     * The test for missing atomic flag is performed here, rather than
> @@ -2938,8 +3086,10 @@ static int cache_grow(struct kmem_cache *cachep,
>
>    cache_init_objs(cachep, slabp);
>
> - if (local_flags&  __GFP_WAIT)
> + if (local_flags&  __GFP_WAIT) {
>     local_irq_disable();
> +  mem_cgroup_kmem_cache_finish_sleep(cachep);
> + }
>    check_irq_off();
>    spin_lock(&l3->list_lock);
>
> @@ -2952,8 +3102,10 @@ static int cache_grow(struct kmem_cache *cachep,
```

> opps1:
>   kmem_freepages(cachep, objp);
> failed:
> - if (local_flags& __GFP_WAIT)
> + if (local_flags& __GFP_WAIT) {
>   local_irq_disable();
> + mem_cgroup_kmem_cache_finish_sleep(cachep);
> + }
>   return 0;
> }
>
> @@ -3712,10 +3864,14 @@ static inline void __cache_free(struct kmem_cache *cachep, void
*objp,
>   */
> void *kmem_cache_alloc(struct kmem_cache *cachep, gfp_t flags)
> {
> - void *ret = __cache_alloc(cachep, flags, __builtin_return_address(0));
> + void *ret;
> +
> + cachep = mem_cgroup_get_kmem_cache(cachep, flags);
> + ret = __cache_alloc(cachep, flags, __builtin_return_address(0));
>
>   trace_kmem_cache_alloc(_RET_IP_, ret,
>   obj_size(cachep), cachep->buffer_size, flags);
> + mem_cgroup_put_kmem_cache(cachep);
>
>   return ret;
> }
> @@ -3727,10 +3883,12 @@ kmem_cache_alloc_trace(size_t size, struct kmem_cache
*cachep, gfp_t flags)
> {
>   void *ret;
>
> + cachep = mem_cgroup_get_kmem_cache(cachep, flags);
>   ret = __cache_alloc(cachep, flags, __builtin_return_address(0));
>
>   trace_kmalloc(_RET_IP_, ret,
>   size, slab_buffer_size(cachep), flags);
> + mem_cgroup_put_kmem_cache(cachep);
>   return ret;

We really need to bypass those when root cgroup is the only user.
Can use static_branch()...

> }
> EXPORT_SYMBOL(kmem_cache_alloc_trace);
> @@ -3739,12 +3897,16 @@ EXPORT_SYMBOL(kmem_cache_alloc_trace);
> #ifdef CONFIG_NUMA

```
>   void *kmem_cache_alloc_node(struct kmem_cache *cachep, gfp_t flags, int nodeid)
>   {
> - void *ret = __cache_alloc_node(cachep, flags, nodeid,
> + void *ret;
> +
> + cachep = mem_cgroup_get_kmem_cache(cachep, flags);
> + ret  = __cache_alloc_node(cachep, flags, nodeid,
>              __builtin_return_address(0));
>
>    trace_kmem_cache_alloc_node(_RET_IP_, ret,
>           obj_size(cachep), cachep->buffer_size,
>           flags, nodeid);
> + mem_cgroup_put_kmem_cache(cachep);
>
>    return ret;
>   }
> @@ -3758,11 +3920,13 @@ void *kmem_cache_alloc_node_trace(size_t size,
>   {
>    void *ret;
>
> + cachep = mem_cgroup_get_kmem_cache(cachep, flags);
>    ret = __cache_alloc_node(cachep, flags, nodeid,
>          __builtin_return_address(0));
>    trace_kmalloc_node(_RET_IP_, ret,
>         size, slab_buffer_size(cachep),
>         flags, nodeid);
> + mem_cgroup_put_kmem_cache(cachep);
>    return ret;
>   }
>   EXPORT_SYMBOL(kmem_cache_alloc_node_trace);
> @@ -3866,9 +4030,35 @@ void kmem_cache_free(struct kmem_cache *cachep, void *objp)
>
>    local_irq_save(flags);
>    debug_check_no_locks_freed(objp, obj_size(cachep));
> +
> +#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
> + {
> +  struct kmem_cache *actual_cachep;
> +
> +  actual_cachep = virt_to_cache(objp);
> +  if (actual_cachep != cachep) {
> +   VM_BUG_ON(actual_cachep->memcg_params.id != -1);
> +   VM_BUG_ON(actual_cachep->memcg_params.orig_cache !=
> +       cachep);
> +   cachep = actual_cachep;
> +  }
> +  /*
> +   * Grab a reference so that the cache is guaranteed to stay
```

> +   * around.
> +   * If we are freeing the last object of a dead memcg cache,
> +   * the kmem_cache_drop_ref() at the end of this function
> +   * will end up freeing the cache.
> +   */
> + kmem_cache_get_ref(cachep);
1) Another obvious candidate to be wrapped by static_branch()...
2) I don't trully follow why we need those references here. Can you
   give us an example of a situation in which the cache can go away?

Also note that we are making a function that used to operate mostly on
local data now issue two atomic operations.

> + }
> +#endif
> +
>   if (!(cachep->flags& SLAB_DEBUG_OBJECTS))
>   debug_check_no_obj_freed(objp, obj_size(cachep));
>   __cache_free(cachep, objp, __builtin_return_address(0));
> +
> + kmem_cache_drop_ref(cachep);
> +
>   local_irq_restore(flags);
>
>   trace_kmem_cache_free(_RET_IP_, objp);
> @@ -3896,9 +4086,19 @@ void kfree(const void *objp)
>   local_irq_save(flags);
>   kfree_debugcheck(objp);
>   c = virt_to_cache(objp);
> +
> + /*
> + * Grab a reference so that the cache is guaranteed to stay around.
> + * If we are freeing the last object of a dead memcg cache, the
> + * kmem_cache_drop_ref() at the end of this function will end up
> + * freeing the cache.
> + */
> + kmem_cache_get_ref(c);
> +
>   debug_check_no_locks_freed(objp, obj_size(c));
>   debug_check_no_obj_freed(objp, obj_size(c));
>   __cache_free(c, (void *)objp, __builtin_return_address(0));
> + kmem_cache_drop_ref(c);
>   local_irq_restore(flags);
>   }
>   EXPORT_SYMBOL(kfree);
> @@ -4167,6 +4367,13 @@ static void cache_reap(struct work_struct *w)
>   list_for_each_entry(searchp,&cache_chain, next) {
>   check_irq_on();

>
> +#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
> + /* For memcg caches, make sure we only reap the active ones. */
> + if (searchp->memcg_params.id == -1&&
> +   !atomic_add_unless(&searchp->refcnt, 1, 0))
> +   continue;
> +#endif
> +
>   /*
>   * We only take the l3 lock if absolutely necessary and we
>   * have established with reasonable certainty that
> @@ -4199,6 +4406,7 @@ static void cache_reap(struct work_struct *w)
>   STATS_ADD_REAPED(searchp, freed);
>   }
>   next:
> + kmem_cache_drop_ref(searchp);
>   cond_resched();
>   }
>   check_irq_on();

Subject: Re: [PATCH v2 07/13] memcg: Slab accounting.
Posted by Suleiman Souhlal on Tue, 13 Mar 2012 22:50:01 GMT
View Forum Message <> Reply to Message

On Sun, Mar 11, 2012 at 3:25 AM, Glauber Costa <glommer@parallels.com> wrote:
> On 03/10/2012 12:39 AM, Suleiman Souhlal wrote:
>> +static inline void
>> +mem_cgroup_kmem_cache_prepare_sleep(struct kmem_cache *cachep)
>> +{
>> +   /*
>> +   * Make sure the cache doesn't get freed while we have interrupts
>> +   * enabled.
>> +   */
>> +   kmem_cache_get_ref(cachep);
>> +   rcu_read_unlock();
>> +}
>
>
> Is this really needed ? After this function call in slab.c, the slab code
> itself accesses cachep a thousand times. If it could be freed, it would
> already explode today for other reasons?
> Am I missing something here?

We need this because once we drop the rcu_read_lock and go to sleep,
the memcg could get deleted, which could lead to the cachep from
getting deleted as well.

So, we need to grab a reference to the cache, to make sure that the
cache doesn't disappear from under us.

>> diff --git a/init/Kconfig b/init/Kconfig
>> index 3f42cd6..e7eb652 100644
>> --- a/init/Kconfig
>> +++ b/init/Kconfig
>> @@ -705,7 +705,7 @@ config CGROUP_MEM_RES_CTLR_SWAP_ENABLED
>> then swapaccount=0 does the trick).
>> config CGROUP_MEM_RES_CTLR_KMEM
>> bool "Memory Resource Controller Kernel Memory accounting
>> (EXPERIMENTAL)"
>> - depends on CGROUP_MEM_RES_CTLR&& EXPERIMENTAL
>> + depends on CGROUP_MEM_RES_CTLR&& EXPERIMENTAL&& !SLOB
>
> Orthogonal question: Will we ever want this (SLOB) ?

I honestly don't know why someone would want to use this and slob at
the same time.
It really doesn't seem like a required feature, in my opinion.
Especially at first.

>> +static struct kmem_cache *
>> +memcg_create_kmem_cache(struct mem_cgroup *memcg, struct kmem_cache
>> *cachep)
>> +{
>> + struct kmem_cache *new_cachep;
>> + struct dentry *dentry;
>> + char *name;
>> + int idx;
>> +
>> + idx = cachep->memcg_params.id;
>> +
>> + dentry = memcg->css.cgroup->dentry;
>> + BUG_ON(dentry == NULL);
>> +
>> + /* Preallocate the space for "dead" at the end */
>> + name = kasprintf(GFP_KERNEL, "%s(%d:%s)dead",
>> + cachep->name, css_id(&memcg->css), dentry->d_name.name);
>> + if (name == NULL)
>> + return cachep;
>> + /* Remove "dead" */
>> + name[strlen(name) - 4] = '\0';
>> +
>> + new_cachep = kmem_cache_create_memcg(cachep, name);
>> +
>> + /*
>> + * Another CPU is creating the same cache?

>> +   * We'll use it next time.
>> +   */
>
> This comment is a bit misleading. Is it really the only reason
> it can fail?
>
> The impression I got is that it can also fail under the normal conditions in
> which kmem_cache_create() fails.

kmem_cache_create() isn't expected to fail often.
I wasn't making an exhaustive lists of why this condition can happen,
just what I think is the most common one is.

>> +/*
>> + * Enqueue the creation of a per-memcg kmem_cache.
>> + * Called with rcu_read_lock.
>> + */
>> +static void
>> +memcg_create_cache_enqueue(struct mem_cgroup *memcg, struct kmem_cache
>> *cachep)
>> +{
>> +   struct create_work *cw;
>> +   unsigned long flags;
>> +
>> +   spin_lock_irqsave(&create_queue_lock, flags);
>
> If we can sleep, why not just create the cache now?
>
> Maybe it would be better to split this in two, and create the cache if
> possible, and a worker if not possible. Then w

That's how I had it in my initial patch, but I was under the
impression that you preferred if we always kicked off the creation to
the workqueue?

Which way do you prefer?

>> @@ -1756,17 +1765,23 @@ static void *kmem_getpages(struct kmem_cache
>> *cachep, gfp_t flags, int nodeid)
>>   if (cachep->flags& SLAB_RECLAIM_ACCOUNT)
>>
>>   flags |= __GFP_RECLAIMABLE;
>>
>> +   nr_pages = (1<< cachep->gfporder);
>> +   if (!mem_cgroup_charge_slab(cachep, flags, nr_pages * PAGE_SIZE))
>> +   return NULL;
>> +
>>   page = alloc_pages_exact_node(nodeid, flags | __GFP_NOTRACK,

>> cachep->gfporder);
>> -   if (!page)
>> +   if (!page) {
>> +   mem_cgroup_uncharge_slab(cachep, nr_pages * PAGE_SIZE);
>>   return NULL;
>> +   }
>
>
>
> Can't the following happen:
>
> *) mem_cgroup_charge_slab() is the first one to touch the slab.
>   Therefore, this first one is billed to root.
> *) A slab is queued for creation.
> *) alloc_pages sleep.
> *) our workers run, and create the cache, therefore filling
>   cachep->memcg_param.memcg
> *) alloc_pages still can't allocate.
> *) uncharge tries to uncharge from cachep->memcg_param.memcg,
>   which doesn't have any charges...
>
> Unless you have a strong oposition to this, to avoid this kind of
> corner cases, we could do what I was doing in the slub:
> Allocate the page first, and then account it.
> (freeing the page if it fails).
>
> I know it is not the way it is done for the user pages, but I believe it to
> be better suited for the slab.

I don't think the situation you're describing can happen, because the
memcg caches get created and selected at the beginning of the slab
allocation, in mem_cgroup_get_kmem_cache() and not in
mem_cgroup_charge_slab(), which is much later.

Once we are in mem_cgroup_charge_slab() we know that the allocation
will be charged to the cgroup.

>> @@ -2269,10 +2288,12 @@ kmem_cache_create (const char *name, size_t size,
>> size_t align,
>>   }
>>
>>   if (!strcmp(pc->name, name)) {
>> -   printk(KERN_ERR
>> -   "kmem_cache_create: duplicate cache %s\n",
>> name);
>> -   dump_stack();
>> -   goto oops;
>> +   if (!memcg) {

```
>> +                        printk(KERN_ERR "kmem_cache_create:
>> duplicate"
>> +                           " cache %s\n", name);
>> +                        dump_stack();
>> +                        goto oops;
>> +                }
>
> Why? Since we are apending the memcg name at the end anyway, duplicates
> still aren't expected.
```

Duplicates can happen if you have hierarchies, because we're only
appending the basename of the cgroup.

```
>> @@ -2703,12 +2787,74 @@ void kmem_cache_destroy(struct kmem_cache *cachep)
>>       if (unlikely(cachep->flags&  SLAB_DESTROY_BY_RCU))
>>
>>               rcu_barrier();
>>
>> +#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
>> +       /* Not a memcg cache */
>> +       if (cachep->memcg_params.id != -1) {
>> +               __clear_bit(cachep->memcg_params.id, cache_types);
>> +               mem_cgroup_flush_cache_create_queue();
>> +       }
>> +#endif
>
>
> This will clear the id when a leaf cache is destroyed. It seems it is not
> what we want, right? We want this id to be cleared only when
> the parent cache is gone.
```

id != -1, for parent caches (that's what the comment is trying to point out).
I will improve the comment.

```
>> +static void
>> +kmem_cache_destroy_work_func(struct work_struct *w)
>> +{
>> +       struct kmem_cache *cachep;
>> +       char *name;
>> +
>> +       spin_lock_irq(&destroy_lock);
>> +       while (!list_empty(&destroyed_caches)) {
>> +               cachep = container_of(list_first_entry(&destroyed_caches,
>> +                   struct mem_cgroup_cache_params, destroyed_list),
>> struct
>> +                   kmem_cache, memcg_params);
>> +               name = (char *)cachep->name;
>> +               list_del(&cachep->memcg_params.destroyed_list);
```

```
>> +            spin_unlock_irq(&destroy_lock);
>> +            synchronize_rcu();
>> +            kmem_cache_destroy(cachep);
>
>            ^^^^^^
>            will destroy the id.
```

See my previous comment.

```
>> @@ -3866,9 +4030,35 @@ void kmem_cache_free(struct kmem_cache *cachep,
>> void *objp)
>>
>>       local_irq_save(flags);
>>       debug_check_no_locks_freed(objp, obj_size(cachep));
>> +
>> +#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
>> +     {
>> +            struct kmem_cache *actual_cachep;
>> +
>> +            actual_cachep = virt_to_cache(objp);
>> +            if (actual_cachep != cachep) {
>> +                  VM_BUG_ON(actual_cachep->memcg_params.id != -1);
>> +                  VM_BUG_ON(actual_cachep->memcg_params.orig_cache
>> !=
>> +                     cachep);
>> +                  cachep = actual_cachep;
>> +            }
>> +            /*
>> +             * Grab a reference so that the cache is guaranteed to
>> stay
>> +             * around.
>> +             * If we are freeing the last object of a dead memcg
>> cache,
>> +             * the kmem_cache_drop_ref() at the end of this function
>> +             * will end up freeing the cache.
>> +             */
>> +            kmem_cache_get_ref(cachep);
>
> 1) Another obvious candidate to be wrapped by static_branch()...
> 2) I don't trully follow why we need those references here. Can you
>   give us an example of a situation in which the cache can go away?
>
> Also note that we are making a function that used to operate mostly on
> local data now issue two atomic operations.
```

Yes, improving this is in my v3 TODO already.

The situation is very simple, and will happen every time we are

freeing the last object of a dead cache.
When we free the last object, kmem_freepages() will drop the last reference, which will cause the kmem_cache to be destroyed right there.
Grabbing an additional reference before freeing the page is just a hack to avoid this situation.

It might be possible to just wrap the free path in rcu_read_lock(), or if that isn't enough, to delay the destruction until the end. I still have to think about this a bit more, to be sure.

Thanks for the detailed review,
-- Suleiman
