Subject: Re: [PATCH 05/10] memcg: Slab accounting.
Posted by Glauber Costa on Tue, 28 Feb 2012 13:24:03 GMT
View Forum Message <> Reply to Message

On 02/27/2012 07:58 PM, Suleiman Souhlal wrote:
> Introduce per-cgroup kmem_caches for memcg slab accounting, that
> get created the first time we do an allocation of that type in the
> cgroup.
> If we are not permitted to sleep in that allocation, the cache
> gets created asynchronously.
And then we allocate from the root cgroup?

> The cgroup cache gets used in subsequent allocations, and permits
> accounting of slab on a per-page basis.
>
> 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.
>
> Allocations that cannot be attributed to a cgroup get charged to
> the root cgroup.
>
> 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.

Since we already track the number of pages in the slab, why do we need a
refcnt?

> Signed-off-by: Suleiman Souhlal<suleiman@google.com>
> ---
> include/linux/memcontrol.h | 30 ++++-
> include/linux/slab.h | 1 +
> include/linux/slab_def.h | 94 ++++++++++++-
> mm/memcontrol.c | 316 ++
> mm/slab.c | 266 ++----
> 5 files changed, 680 insertions(+), 27 deletions(-)
>
> diff --git a/include/linux/memcontrol.h b/include/linux/memcontrol.h
> index 4d34356..f5458b0 100644
> --- a/include/linux/memcontrol.h
> +++ b/include/linux/memcontrol.h
> @@ -421,13 +421,41 @@ 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..fe21a91 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 0x00008000UL /* memcg kmem_cache */
>   #define SLAB_STORE_USER 0x00010000UL /* DEBUG: Store the last owner for bug hunting

*/
> #define SLAB_PANIC 0x00040000UL /* Panic if kmem_cache_create() fails */
> /*

We'll get to this later, but I dislike adding this flag, since we can
just test for existence of a pointer that we need to track anyway in
the slab structure.

This may create some problems when we track it for root memcg, but this
is something your patchset does, and I believe we shouldn't.

> diff --git a/include/linux/slab_def.h b/include/linux/slab_def.h
> index fbd1117..449a0de 100644
> --- a/include/linux/slab_def.h
> +++ b/include/linux/slab_def.h
> @@ -41,6 +41,10 @@ struct kmem_cache {
>   /* force GFP flags, e.g. GFP_DMA */
>   gfp_t gfpflags;
>
> +#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
> + int id;   /* id used for slab accounting */
> +#endif
> +
What role does it play? Is it the same as the array index in my patchset?

>   size_t colour;   /* cache colouring range */
>   unsigned int colour_off; /* colour offset */
>   struct kmem_cache *slabp_cache;
> @@ -51,7 +55,7 @@ struct kmem_cache {
>   void (*ctor)(void *obj);
>
> /* 4) cache creation/removal */
> - const char *name;
> + char *name;
>   struct list_head next;
>
> /* 5) statistics */
> @@ -78,9 +82,26 @@ struct kmem_cache {
>   * variables contain the offset to the user object and its size.
>   */
>   int obj_offset;
> - int obj_size;
> #endif /* CONFIG_DEBUG_SLAB */
>
> +#if defined(CONFIG_DEBUG_SLAB) ||
defined(CONFIG_CGROUP_MEM_RES_CTLR_KMEM)
> + int obj_size;
> +#endif

> +
> +#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
> + /* Original cache parameters, used when creating a memcg cache */
> + size_t orig_align;
> + unsigned long orig_flags;
> +
> + struct mem_cgroup *memcg;
> +
> + /* Who we copied from when creating cpuset cache */
> + struct kmem_cache *orig_cache;
> +
> + atomic_t refcnt;
> + struct list_head destroyed_list; /* Used when deleting cpuset cache */
> +#endif /* CONFIG_CGROUP_MEM_RES_CTLR_KMEM */
I think you're adding way to many things here.

I prefer the approach I took of having a memcg pointer, and then having
that stuff into memcg. It will be better for people not interested in
this feature - like if you compile this in, but then does not mount memcg.

> +
> /* 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
> @@ -212,4 +233,73 @@ found:
>
> #endif /* CONFIG_NUMA */
>
> +#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
> +
> +#define MAX_KMEM_CACHE_TYPES 300
> +
> +struct kmem_cache *kmem_cache_create_memcg(struct kmem_cache *cachep,
> + char *name);
> +void kmem_cache_destroy_cpuset(struct kmem_cache *cachep);
> +void kmem_cache_drop_ref(struct kmem_cache *cachep);
> +
> +static inline void
> +kmem_cache_get_ref(struct kmem_cache *cachep)
> +{
> + if ((cachep->flags& SLAB_MEMCG)&&
> + 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();
> +}
> +
> +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/mm/memcontrol.c b/mm/memcontrol.c
> index c82ca1c..d1c0cd7 100644
> --- a/mm/memcontrol.c
> +++ b/mm/memcontrol.c
> @@ -297,6 +297,11 @@ struct mem_cgroup {
> #ifdef CONFIG_INET
>   struct tcp_memcontrol tcp_mem;
> #endif
> +
> +#if defined(CONFIG_CGROUP_MEM_RES_CTLR_KMEM)&& defined(CONFIG_SLAB)
> + /* Slab accounting */
> + struct kmem_cache *slabs[MAX_KMEM_CACHE_TYPES];
> +#endif
>   int independent_kmem_limit;
> };
>
> @@ -5633,6 +5638,312 @@ memcg_uncharge_kmem(struct mem_cgroup *memcg, long long
delta)
>   res_counter_uncharge(&memcg->res, delta);
> }
>
> +#ifdef CONFIG_SLAB

Why CONFIG_SLAB? If this is in memcontrol.c, shouldn't have anything
slab-specific here...

> +static struct kmem_cache *
> +memcg_create_kmem_cache(struct mem_cgroup *memcg, int idx,
> +   struct kmem_cache *cachep, gfp_t gfp)
> +{
> + struct kmem_cache *new_cachep;
> + struct dentry *dentry;
> + char *name;
> + int len;
> +
> + if ((gfp& GFP_KERNEL) != GFP_KERNEL)
> + return cachep;
> +
> + dentry = memcg->css.cgroup->dentry;
> + BUG_ON(dentry == NULL);
> + len = strlen(cachep->name);
> + len += dentry->d_name.len;
> + len += 7; /* Space for "()", NUL and appending "dead" */
> + name = kmalloc(len, GFP_KERNEL | __GFP_NOACCOUNT);
> +
> + if (name == NULL)

```
> +  return cachep;
> +
> + snprintf(name, len, "%s(%s)", cachep->name,
> +     dentry ? (const char *)dentry->d_name.name : "/");
> + name[len - 5] = '\0'; /* Make sure we can append "dead" later */
> +
> + new_cachep = kmem_cache_create_memcg(cachep, name);
> +
> + /*
> +  * Another CPU is creating the same cache?
> +  * We'll use it next time.
> +  */
> + if (new_cachep == NULL) {
> +  kfree(name);
> +  return cachep;
> + }
> +
> + new_cachep->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);
> +
> +/*
> + * 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->id,
> +      cw->cachep, GFP_KERNEL);
> +  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);
> +  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);
> +
> +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);
> + 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);
> +
> + cw = kmalloc(sizeof(struct create_work), GFP_NOWAIT | __GFP_NOACCOUNT);
> + if (cw == NULL)
> +  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 kmem_cache *ret;
> + struct mem_cgroup *memcg;
> + int idx;
> +
> + rcu_read_lock();
> +
> + if (in_interrupt())
> +  return cachep;
> + if (current == NULL)
> +  return cachep;
> +
```

```
> + gfp |= cachep->gfpflags;
> + if ((gfp&  __GFP_NOACCOUNT) || (gfp&  __GFP_NOFAIL))
> +  return cachep;
> +
> + if (cachep->flags&  SLAB_MEMCG)
> +  return cachep;
> +
> + memcg = mem_cgroup_from_task(current);
> + idx = cachep->id;
> +
> + if (memcg == NULL || memcg == root_mem_cgroup)
> +  return cachep;
> +
> + VM_BUG_ON(idx == -1);
> +
> + if (rcu_access_pointer(memcg->slabs[idx]) == NULL) {
> +  if ((gfp&  GFP_KERNEL) == GFP_KERNEL) {
> +   if (!css_tryget(&memcg->css))
> +    return cachep;
> +   rcu_read_unlock();
> +   ret = memcg_create_kmem_cache(memcg, idx, cachep, gfp);
> +   rcu_read_lock();
> +   css_put(&memcg->css);
> +   return ret;
> +  } else {
> +   /*
> +    * Not a 'normal' slab allocation, so just enqueue
> +    * the creation of the memcg cache and let the current
> +    * allocation use the normal cache.
> +    */
> +   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->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();
> + if (cachep->flags&  SLAB_MEMCG)
> +  memcg = cachep->memcg;
> + else
> +  memcg = NULL;
> +
> + if (memcg&&  !css_tryget(&memcg->css))
> +  memcg = NULL;
> + rcu_read_unlock();
> +
> + 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();
> + if (cachep->flags&  SLAB_MEMCG)
> +  memcg = cachep->memcg;
> + else
> +  memcg = NULL;
> +
> + 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 = 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);
> +  }
> + }
> +}
> +#else /* CONFIG_SLAB */
> +static void
> +memcg_slab_init(struct mem_cgroup *memcg)
> +{
> +}
> +
> +static void
> +memcg_slab_move(struct mem_cgroup *memcg)
> +{
> +}
> +#endif /* CONFIG_SLAB */
> +
>  static void
>  memcg_kmem_init(struct mem_cgroup *memcg, struct mem_cgroup *parent)
>  {
```

> @@ -5642,6 +5953,9 @@ memcg_kmem_init(struct mem_cgroup *memcg, struct mem_cgroup
*parent)
> if (parent&& parent != root_mem_cgroup)
> parent_res =&parent->kmem_bytes;
> res_counter_init(&memcg->kmem_bytes, parent_res);
> +
> + memcg_slab_init(memcg);
> +
> memcg->independent_kmem_limit = 0;
> }
>
> @@ -5651,6 +5965,8 @@ memcg_kmem_move(struct mem_cgroup *memcg)
> unsigned long flags;
> long kmem_bytes;
>
> + memcg_slab_move(memcg);
> +
> spin_lock_irqsave(&memcg->kmem_bytes.lock, flags);
> kmem_bytes = memcg->kmem_bytes.usage;
> res_counter_uncharge_locked(&memcg->kmem_bytes, kmem_bytes);
> diff --git a/mm/slab.c b/mm/slab.c
> index f0bd785..6c6bc49 100644
> --- a/mm/slab.c
> +++ b/mm/slab.c
> @@ -301,6 +301,8 @@ 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 +328,11 @@ 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 +1763,23 @@ 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;
> + }
>
> - 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 +1812,8 @@ 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 +2239,17 @@ 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 +2266,6 @@ 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 +2286,12 @@ 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;
> +   }
>     }
>    }
>
> @@ -2359,9 +2378,9 @@ kmem_cache_create (const char *name, size_t size, size_t align,
>    align = ralign;
>
>    if (slab_is_available())
> -  gfp = GFP_KERNEL;
> +  gfp = GFP_KERNEL | __GFP_NOACCOUNT;
>    else
> -  gfp = GFP_NOWAIT;
> +  gfp = GFP_NOWAIT | __GFP_NOACCOUNT;
>
>    /* Get cache's description obj. */
>    cachep = kmem_cache_zalloc(&cache_cache, gfp);
> @@ -2369,9 +2388,15 @@ kmem_cache_create (const char *name, size_t size, size_t align,
```

```
>     goto oops;
>
>    cachep->nodelists = (struct kmem_list3 **)&cachep->array[nr_cpu_ids];
> -#if DEBUG
> +
> +#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
>    cachep->obj_size = size;
> + cachep->orig_align = orig_align;
> + cachep->orig_flags = orig_flags;
> +#endif
>
> +#if DEBUG
> + cachep->obj_size = size;
>    /*
>     * Both debugging options require word-alignment which is calculated
>     * into align above.
> @@ -2477,7 +2502,23 @@ 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->orig_cache = NULL;
> + atomic_set(&cachep->refcnt, 1);
> + INIT_LIST_HEAD(&cachep->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->id = id;
> + } else
> +  cachep->id = -1;
> +#endif
>
>    if (setup_cpu_cache(cachep, gfp)) {
>     __kmem_cache_destroy(cachep);
> @@ -2502,13 +2543,54 @@ 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->orig_flags&  ~SLAB_PANIC;
> + mutex_lock(&cache_chain_mutex);
> + new = __kmem_cache_create(name, cachep->obj_size, cachep->orig_align,
> +     flags, cachep->ctor, 1);
> + if (new == NULL) {
> +  mutex_unlock(&cache_chain_mutex);
> +  return NULL;
> + }
> + new->flags |= SLAB_MEMCG;
> + new->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 | __GFP_NOACCOUNT);
> + mutex_unlock(&cache_chain_mutex);
> +
> + return new;
> +}
> +#endif /* CONFIG_CGROUP_MEM_RES_CTLR_KMEM */
> +
>   #if DEBUG
>   static void check_irq_off(void)
```

```
>   {
> @@ -2703,12 +2785,73 @@ 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->id != -1) {
> +  __clear_bit(cachep->id, cache_types);
> +  mem_cgroup_flush_cache_create_queue();
> + }
> +#endif
>     __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 = list_first_entry(&destroyed_caches, struct kmem_cache,
> +       destroyed_list);
> +  name = (char *)cachep->name;
> +  list_del(&cachep->destroyed_list);
> +  spin_unlock_irq(&destroy_lock);
> +  synchronize_rcu();
> +  kmem_cache_destroy(cachep);
> +  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->flags& SLAB_MEMCG));
> +
> + /*
> + * 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->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->flags& SLAB_MEMCG)&&
> + 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 +3051,10 @@ 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
> @@ -2920,6 +3065,13 @@ static int cache_grow(struct kmem_cache *cachep,
> kmem_flagcheck(cachep, flags);
>
> /*
> + * alloc_slabmgmt() might invoke the slab allocator itself, so
> + * make sure we don't recurse in slab accounting.
> + */
> + if (flags& __GFP_NOACCOUNT)
> + local_flags |= __GFP_NOACCOUNT;

```
> +
> + /*
>     * Get mem for the objs.  Attempt to allocate a physical page from
>     * 'nodeid'.
>     */
> @@ -2938,8 +3090,10 @@ 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 +3106,10 @@ 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 +3868,14 @@ 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 +3887,12 @@ 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;
>  }
>  EXPORT_SYMBOL(kmem_cache_alloc_trace);
> @@ -3739,12 +3901,16 @@ 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 +3924,13 @@ 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);
> @@ -3772,11 +3940,16 @@ static __always_inline void *
>  __do_kmalloc_node(size_t size, gfp_t flags, int node, void *caller)
>  {
>   struct kmem_cache *cachep;
```

```
> + void *ret;
>
>    cachep = kmem_find_general_cachep(size, flags);
>    if (unlikely(ZERO_OR_NULL_PTR(cachep)))
>     return cachep;
> - return kmem_cache_alloc_node_trace(size, cachep, flags, node);
> + cachep = mem_cgroup_get_kmem_cache(cachep, flags);
> + ret = kmem_cache_alloc_node_trace(size, cachep, flags, node);
> + mem_cgroup_put_kmem_cache(cachep);
> +
> + return ret;
>   }
>
>   #if defined(CONFIG_DEBUG_SLAB) || defined(CONFIG_TRACING)
> @@ -3822,10 +3995,12 @@ static __always_inline void *__do_kmalloc(size_t size, gfp_t flags,
>    cachep = __find_general_cachep(size, flags);
>    if (unlikely(ZERO_OR_NULL_PTR(cachep)))
>     return cachep;
> + cachep = mem_cgroup_get_kmem_cache(cachep, flags);
>    ret = __cache_alloc(cachep, flags, caller);
>
>    trace_kmalloc((unsigned long) caller, ret,
>           size, cachep->buffer_size, flags);
> + mem_cgroup_put_kmem_cache(cachep);
>
>    return ret;
>   }
> @@ -3866,9 +4041,34 @@ 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->flags&  SLAB_MEMCG));
> +   VM_BUG_ON(actual_cachep->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);
> + }
> +#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 +4096,19 @@ 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 +4377,13 @@ 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->flags&  SLAB_MEMCG)&&
> +      !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 +4416,7 @@ static void cache_reap(struct work_struct *w)
> STATS_ADD_REAPED(searchp, freed);
> }
> next:
> + kmem_cache_drop_ref(searchp);
> cond_resched();
> }
> check_irq_on();
> @@ -4412,8 +4630,8 @@ static ssize_t slabinfo_write(struct file *file, const char __user
*buffer,
> res = 0;
> } else {
> res = do_tune_cpucache(cachep, limit,
> - batchcount, shared,
> - GFP_KERNEL);
> + batchcount, shared, GFP_KERNEL |
> + __GFP_NOACCOUNT);
> }
> break;
> }

Subject: Re: [PATCH 05/10] memcg: Slab accounting.
Posted by Suleiman Souhlal on Tue, 28 Feb 2012 23:31:14 GMT
View Forum Message <> Reply to Message

On Tue, Feb 28, 2012 at 5:24 AM, Glauber Costa <glommer@parallels.com> wrote:
> On 02/27/2012 07:58 PM, Suleiman Souhlal wrote:
>>
>> Introduce per-cgroup kmem_caches for memcg slab accounting, that
>> get created the first time we do an allocation of that type in the
>> cgroup.
>> If we are not permitted to sleep in that allocation, the cache
>> gets created asynchronously.
>
> And then we allocate from the root cgroup?

Yes, the allocation will go to the root cgroup (or not get accounted
at all if you don't have CONFIG_CGROUP_MEM_RES_CTLR_KMEM_ACCT_ROOT).
Once the workqueue runs and creates the memcg cache, all the
allocations of that type will start using it.

>> The cgroup cache gets used in subsequent allocations, and permits
>> accounting of slab on a per-page basis.
>>
>> 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.
>>
>> Allocations that cannot be attributed to a cgroup get charged to
>> the root cgroup.
>>
>> 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.
>
>
> Since we already track the number of pages in the slab, why do we need a
> refcnt?

I must be missing something, but I don't see a counter of the number
of active pages in the cache in the code. :-(

>> diff --git a/include/linux/slab.h b/include/linux/slab.h
>> index 573c809..fe21a91 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   0x00008000UL   /* memcg kmem_cache */
>> #define SLAB_STORE_USER   0x00010000UL   /* DEBUG: Store the
>> last owner for bug hunting */
>> #define SLAB_PANIC   0x00040000UL   /* Panic if
>> kmem_cache_create() fails */
>> /*
>
>
> We'll get to this later, but I dislike adding this flag, since we can just
> test for existence of a pointer that we need to track anyway in
> the slab structure.

I might be able to remove this flag. I'll try to get that done in v2.

>
> This may create some problems when we track it for root memcg, but this is
> something your patchset does, and I believe we shouldn't.
>
>
>> diff --git a/include/linux/slab_def.h b/include/linux/slab_def.h
>> index fbd1117..449a0de 100644
>> --- a/include/linux/slab_def.h

>> +++ b/include/linux/slab_def.h
>> @@ -41,6 +41,10 @@ struct kmem_cache {
>> /* force GFP flags, e.g. GFP_DMA */
>> gfp_t gfpflags;
>>
>> +#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
>> + int id; /* id used for slab accounting */
>> +#endif
>> +
>
> What role does it play? Is it the same as the array index in my patchset?

Yes, this is the index into the memcg slab array.
The id gets allocated when someone does kmem_cache_create().

>> size_t colour; /* cache colouring range */
>> unsigned int colour_off; /* colour offset */
>> struct kmem_cache *slabp_cache;
>> @@ -51,7 +55,7 @@ struct kmem_cache {
>> void (*ctor)(void *obj);
>>
>> /* 4) cache creation/removal */
>> - const char *name;
>> + char *name;
>> struct list_head next;
>>
>> /* 5) statistics */
>> @@ -78,9 +82,26 @@ struct kmem_cache {
>> * variables contain the offset to the user object and its size.
>> */
>> int obj_offset;
>> - int obj_size;
>> #endif /* CONFIG_DEBUG_SLAB */
>>
>> +#if defined(CONFIG_DEBUG_SLAB) ||
>> defined(CONFIG_CGROUP_MEM_RES_CTLR_KMEM)
>> + int obj_size;
>> +#endif
>> +
>> +#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
>> + /* Original cache parameters, used when creating a memcg cache */
>> + size_t orig_align;
>> + unsigned long orig_flags;
>> +
>> + struct mem_cgroup *memcg;
>> +
>> + /* Who we copied from when creating cpuset cache */

>> +	struct kmem_cache *orig_cache;
>> +
>> +	atomic_t refcnt;
>> +	struct list_head destroyed_list; /* Used when deleting cpuset
>> cache */
>> +#endif /* CONFIG_CGROUP_MEM_RES_CTLR_KMEM */
>
> I think you're adding way to many things here.
>
> I prefer the approach I took of having a memcg pointer, and then having that
> stuff into memcg. It will be better for people not interested in this
> feature - like if you compile this in, but then does not mount memcg.

Given that there are only on the order of a hundred different
kmem_caches, when slab accounting is disabled, I'm not sure the 52
bytes (or 64?) that are being added here are a big concern.

If you really think this is important, I can move them to a different structure.

>> diff --git a/mm/memcontrol.c b/mm/memcontrol.c
>> index c82ca1c..d1c0cd7 100644
>> --- a/mm/memcontrol.c
>> +++ b/mm/memcontrol.c
>> @@ -297,6 +297,11 @@ struct mem_cgroup {
>> #ifdef CONFIG_INET
>> 	struct tcp_memcontrol tcp_mem;
>> #endif
>> +
>> +#if defined(CONFIG_CGROUP_MEM_RES_CTLR_KMEM)&& defined(CONFIG_SLAB)
>>
>> +	/* Slab accounting */
>> +	struct kmem_cache *slabs[MAX_KMEM_CACHE_TYPES];
>> +#endif
>> 	int independent_kmem_limit;
>> };
>>
>> @@ -5633,6 +5638,312 @@ memcg_uncharge_kmem(struct mem_cgroup *memcg, long
>> long delta)
>> 		res_counter_uncharge(&memcg->res, delta);
>> }
>>
>> +#ifdef CONFIG_SLAB
>
>
> Why CONFIG_SLAB? If this is in memcontrol.c, shouldn't have anything
> slab-specific here...

I'm not sure this code will compile with another slab allocator.

I'll look into what I need to do get rid of these #ifdefs.

Thanks,
-- Suleiman
