Subject: Re: [PATCH v2 18/29] memcg: kmem controller charge/uncharge infrastructure
Posted by KAMEZAWA Hiroyuki on Tue, 15 May 2012 02:57:35 GMT
View Forum Message <> Reply to Message

(2012/05/12 2:44), Glauber Costa wrote:

> With all the dependencies already in place, this patch introduces
> the charge/uncharge functions for the slab cache accounting in memcg.
>
> Before we can charge a cache, we need to select the right cache.
> This is done by using the function __mem_cgroup_get_kmem_cache().
>
> If we should use the root kmem cache, this function tries to detect
> that and return as early as possible.
>
> The charge and uncharge functions comes in two flavours:
> * __mem_cgroup_(un)charge_slab(), that assumes the allocation is
> a slab page, and
> * __mem_cgroup_(un)charge_kmem(), that does not. This later exists
> because the slub allocator draws the larger kmalloc allocations
> from the page allocator.
>
> In memcontrol.h those functions are wrapped in inline acessors.
> The idea is to later on, patch those with jump labels, so we don't
> incur any overhead when no mem cgroups are being used.
>
> Because the slub allocator tends to inline the allocations whenever
> it can, those functions need to be exported so modules can make use
> of it properly.
>
> I apologize in advance to the reviewers. This patch is quite big, but
> I was not able to split it any further due to all the dependencies
> between the code.
>
> This code is inspired by the code written by Suleiman Souhlal,
> but heavily changed.
>
> Signed-off-by: Glauber Costa <glommer@parallels.com>
> CC: Christoph Lameter <cl@linux.com>
> CC: Pekka Enberg <penberg@cs.helsinki.fi>
> CC: Michal Hocko <mhocko@suse.cz>
> CC: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
> CC: Johannes Weiner <hannes@cmpxchg.org>
> CC: Suleiman Souhlal <suleiman@google.com>
> ---
> include/linux/memcontrol.h | 67 ++++++++
> init/Kconfig | 2 +-

> mm/memcontrol.c | 379 ++-
> 3 files changed, 446 insertions(+), 2 deletions(-)
>
> diff --git a/include/linux/memcontrol.h b/include/linux/memcontrol.h
> index f93021a..c555799 100644
> --- a/include/linux/memcontrol.h
> +++ b/include/linux/memcontrol.h
> @@ -21,6 +21,7 @@
> #define _LINUX_MEMCONTROL_H
> #include <linux/cgroup.h>
> #include <linux/vm_event_item.h>
> +#include <linux/hardirq.h>
>
> struct mem_cgroup;
> struct page_cgroup;
> @@ -447,6 +448,19 @@ void mem_cgroup_register_cache(struct mem_cgroup *memcg,
> void mem_cgroup_release_cache(struct kmem_cache *cachep);
> extern char *mem_cgroup_cache_name(struct mem_cgroup *memcg,
>   struct kmem_cache *cachep);
> +
> +void mem_cgroup_flush_cache_create_queue(void);
> +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);
> +
> +bool __mem_cgroup_new_kmem_page(struct page *page, gfp_t gfp);
> +void __mem_cgroup_free_kmem_page(struct page *page);
> +
> +struct kmem_cache *
> +__mem_cgroup_get_kmem_cache(struct kmem_cache *cachep, gfp_t gfp);
> +
> +#define mem_cgroup_kmem_on 1
> #else
> static inline void mem_cgroup_register_cache(struct mem_cgroup *memcg,
>   struct kmem_cache *s)
> @@ -463,6 +477,59 @@ static inline void sock_update_memcg(struct sock *sk)
> static inline void sock_release_memcg(struct sock *sk)
> {
> }
> +
> +static inline void
> +mem_cgroup_flush_cache_create_queue(void)
> +{
> +}
> +
> +static inline void mem_cgroup_destroy_cache(struct kmem_cache *cachep)
> +{
> +}

```
> +
> +#define mem_cgroup_kmem_on 0
> +#define __mem_cgroup_get_kmem_cache(a, b) a
> +#define __mem_cgroup_charge_slab(a, b, c) false
> +#define __mem_cgroup_new_kmem_page(a, gfp) false
> +#define __mem_cgroup_uncharge_slab(a, b)
> +#define __mem_cgroup_free_kmem_page(b)
>  #endif /* CONFIG_CGROUP_MEM_RES_CTLR_KMEM */
> +static __always_inline struct kmem_cache *
> +mem_cgroup_get_kmem_cache(struct kmem_cache *cachep, gfp_t gfp)
> +{
> + if (mem_cgroup_kmem_on && current->mm && !in_interrupt())
> +  return __mem_cgroup_get_kmem_cache(cachep, gfp);
> + return cachep;
> +}
> +
> +static __always_inline bool
> +mem_cgroup_charge_slab(struct kmem_cache *cachep, gfp_t gfp, size_t size)
> +{
> + if (mem_cgroup_kmem_on)
> +  return __mem_cgroup_charge_slab(cachep, gfp, size);
> + return true;
> +}
> +
> +static __always_inline void
> +mem_cgroup_uncharge_slab(struct kmem_cache *cachep, size_t size)
> +{
> + if (mem_cgroup_kmem_on)
> +  __mem_cgroup_uncharge_slab(cachep, size);
> +}
> +
> +static __always_inline
> +bool mem_cgroup_new_kmem_page(struct page *page, gfp_t gfp)
> +{
> + if (mem_cgroup_kmem_on && current->mm && !in_interrupt())
> +  return __mem_cgroup_new_kmem_page(page, gfp);
> + return true;
> +}
> +
> +static __always_inline
> +void mem_cgroup_free_kmem_page(struct page *page)
> +{
> + if (mem_cgroup_kmem_on)
> +  __mem_cgroup_free_kmem_page(page);
> +}
>  #endif /* _LINUX_MEMCONTROL_H */
>
> diff --git a/init/Kconfig b/init/Kconfig
```

> index 72f33fa..071b7e3 100644
> --- a/init/Kconfig
> +++ b/init/Kconfig
> @@ -696,7 +696,7 @@ 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
> default n
> help
> The Kernel Memory extension for Memory Resource Controller can limit
> diff --git a/mm/memcontrol.c b/mm/memcontrol.c
> index a8171cb..5a7416b 100644
> --- a/mm/memcontrol.c
> +++ b/mm/memcontrol.c
> @@ -10,6 +10,10 @@
> * Copyright (C) 2009 Nokia Corporation
> * Author: Kirill A. Shutemov
> *
> + * Kernel Memory Controller
> + * Copyright (C) 2012 Parallels Inc. and Google Inc.
> + * Authors: Glauber Costa and Suleiman Souhlal
> + *
> * This program is free software; you can redistribute it and/or modify
> * it under the terms of the GNU General Public License as published by
> * the Free Software Foundation; either version 2 of the License, or
> @@ -321,6 +325,11 @@ 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
> };
>
> int memcg_css_id(struct mem_cgroup *memcg)
> @@ -414,6 +423,9 @@ static void mem_cgroup_put(struct mem_cgroup *memcg);
> #include <net/ip.h>
>
> static bool mem_cgroup_is_root(struct mem_cgroup *memcg);
> +static int memcg_charge_kmem(struct mem_cgroup *memcg, gfp_t gfp, s64 delta);
> +static void memcg_uncharge_kmem(struct mem_cgroup *memcg, s64 delta);
> +
> void sock_update_memcg(struct sock *sk)
> {

```
>   if (mem_cgroup_sockets_enabled) {
> @@ -484,7 +496,14 @@ char *mem_cgroup_cache_name(struct mem_cgroup *memcg, struct
kmem_cache *cachep)
>   return name;
>  }
>
> +static inline bool mem_cgroup_kmem_enabled(struct mem_cgroup *memcg)
> +{
> + return !mem_cgroup_disabled() && memcg &&
> +        !mem_cgroup_is_root(memcg) && memcg->kmem_accounted;
> +}
> +
>  struct ida cache_types;
> +static DEFINE_MUTEX(memcg_cache_mutex);
>
>  void mem_cgroup_register_cache(struct mem_cgroup *memcg,
>           struct kmem_cache *cachep)
> @@ -504,6 +523,298 @@ void mem_cgroup_release_cache(struct kmem_cache *cachep)
>   if (cachep->memcg_params.id != -1)
>    ida_simple_remove(&cache_types, cachep->memcg_params.id);
>  }
> +
> +
> +static struct kmem_cache *memcg_create_kmem_cache(struct mem_cgroup *memcg,
> +        struct kmem_cache *cachep)
> +{
> + struct kmem_cache *new_cachep;
> + int idx;
> +
> + BUG_ON(!mem_cgroup_kmem_enabled(memcg));
> +
> + idx = cachep->memcg_params.id;
> +
> + mutex_lock(&memcg_cache_mutex);
> + new_cachep = memcg->slabs[idx];
> + if (new_cachep)
> +  goto out;
> +
> + new_cachep = kmem_cache_dup(memcg, cachep);
> +
> + if (new_cachep == NULL) {
> +  new_cachep = cachep;
> +  goto out;
> + }
> +
> + mem_cgroup_get(memcg);
> + memcg->slabs[idx] = new_cachep;
> + new_cachep->memcg_params.memcg = memcg;
```

```
> + atomic_set(&new_cachep->memcg_params.refcnt, 1);
> +out:
> + mutex_unlock(&memcg_cache_mutex);
> + return new_cachep;
> +}
> +
> +struct create_work {
> + struct mem_cgroup *memcg;
> + struct kmem_cache *cachep;
> + struct list_head list;
> +};
> +
> +/* Use a single spinlock for destruction and creation, not a frequent op */
> +static DEFINE_SPINLOCK(cache_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(&cache_queue_lock, flags);
> + list_for_each_entry_safe(cw, tmp, &create_queue, list) {
> +  list_del(&cw->list);
> +  kfree(cw);
> + }
> + spin_unlock_irqrestore(&cache_queue_lock, flags);
> +}
> +
> +static void memcg_create_cache_work_func(struct work_struct *w)
> +{
> + struct create_work *cw, *tmp;
> + unsigned long flags;
> + LIST_HEAD(create_unlocked);
> +
> + spin_lock_irqsave(&cache_queue_lock, flags);
> + list_for_each_entry_safe(cw, tmp, &create_queue, list)
> +  list_move(&cw->list, &create_unlocked);
> + spin_unlock_irqrestore(&cache_queue_lock, flags);
> +
> + list_for_each_entry_safe(cw, tmp, &create_unlocked, list) {
```

```
> +  list_del(&cw->list);
> +  memcg_create_kmem_cache(cw->memcg, cw->cachep);
> +  /* Drop the reference gotten when we enqueued. */
> +  css_put(&cw->memcg->css);
> +  kfree(cw);
> + }
> +}
> +
> +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(&cache_queue_lock, flags);
> + list_for_each_entry(cw, &create_queue, list) {
> +  if (cw->memcg == memcg && cw->cachep == cachep) {
> +   spin_unlock_irqrestore(&cache_queue_lock, flags);
> +   return;
> +  }
> + }
> + spin_unlock_irqrestore(&cache_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(&cache_queue_lock, flags);
> + list_add_tail(&cw->list, &create_queue);
> + spin_unlock_irqrestore(&cache_queue_lock, flags);
> +
> + schedule_work(&memcg_create_cache_work);
> +}
> +
```

```
> +/*
> + * Return the kmem_cache we're supposed to use for a slab allocation.
> + * We 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.
> + *
> + * Can't be called in interrupt context or from kernel threads.
> + * This function needs to be called 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;
> + struct task_struct *p;
> +
> + gfp |=  cachep->allocflags;
> +
> + if (cachep->memcg_params.memcg)
> +  return cachep;
> +
> + idx = cachep->memcg_params.id;
> + VM_BUG_ON(idx == -1);
> +
> + p = rcu_dereference(current->mm->owner);
> + memcg = mem_cgroup_from_task(p);
> +
> + if (!mem_cgroup_kmem_enabled(memcg))
> +  return cachep;
> +
> + if (memcg->slabs[idx] == NULL) {
> +  memcg_create_cache_enqueue(memcg, cachep);
> +  return cachep;
> + }
> +
> + return memcg->slabs[idx];
> +}
> +EXPORT_SYMBOL(__mem_cgroup_get_kmem_cache);
> +
> +bool __mem_cgroup_new_kmem_page(struct page *page, gfp_t gfp)
> +{
> + struct mem_cgroup *memcg;
> + struct page_cgroup *pc;
> + bool ret = true;
```

```
> + size_t size;
> + struct task_struct *p;
> +
> + if (!current->mm || in_interrupt())
> +  return true;
> +
> + rcu_read_lock();
> + p = rcu_dereference(current->mm->owner);
> + memcg = mem_cgroup_from_task(p);
> +
> + if (!mem_cgroup_kmem_enabled(memcg))
> +  goto out;
> +
> + mem_cgroup_get(memcg);
> +
> + size = (1 << compound_order(page)) << PAGE_SHIFT;
> +
> + ret = memcg_charge_kmem(memcg, gfp, size) == 0;
> + if (!ret) {
> +  mem_cgroup_put(memcg);
> +  goto out;
> + }
> +
> + pc = lookup_page_cgroup(page);
> + lock_page_cgroup(pc);
> + pc->mem_cgroup = memcg;
> + SetPageCgroupUsed(pc);
> + unlock_page_cgroup(pc);
> +
> +out:
> + rcu_read_unlock();
> + return ret;
> +}
> +EXPORT_SYMBOL(__mem_cgroup_new_kmem_page);
> +
> +void __mem_cgroup_free_kmem_page(struct page *page)
> +{
> + struct mem_cgroup *memcg;
> + size_t size;
> + struct page_cgroup *pc;
> +
> + if (mem_cgroup_disabled())
> +  return;
> +
> + pc = lookup_page_cgroup(page);
> + lock_page_cgroup(pc);
> + memcg = pc->mem_cgroup;
> + pc->mem_cgroup = NULL;
```

```
> + if (!PageCgroupUsed(pc)) {
> +  unlock_page_cgroup(pc);
> +  return;
> + }
> + ClearPageCgroupUsed(pc);
> + unlock_page_cgroup(pc);
> +
> + /*
> +  * The classical disabled check won't work
> +  * for uncharge, since it is possible that the user enabled
> +  * kmem tracking, allocated, and then disabled.
> +  *
> +  * We trust if there is a memcg associated with the page,
> +  * it is a valid allocation
> +  */
> +
> + if (!memcg)
> +  return;
> +
> + WARN_ON(mem_cgroup_is_root(memcg));
> + size = (1 << compound_order(page)) << PAGE_SHIFT;
> + memcg_uncharge_kmem(memcg, size);
> + mem_cgroup_put(memcg);
> +}
> +EXPORT_SYMBOL(__mem_cgroup_free_kmem_page);
> +
> +bool __mem_cgroup_charge_slab(struct kmem_cache *cachep, gfp_t gfp, size_t size)
> +{
> + struct mem_cgroup *memcg;
> + bool ret = true;
> +
> + rcu_read_lock();
> + memcg = cachep->memcg_params.memcg;
> + if (!mem_cgroup_kmem_enabled(memcg))
> +  goto out;
> +
> + ret = memcg_charge_kmem(memcg, gfp, size) == 0;
> +out:
> + rcu_read_unlock();
> + return ret;
> +}
> +EXPORT_SYMBOL(__mem_cgroup_charge_slab);
> +
> +void __mem_cgroup_uncharge_slab(struct kmem_cache *cachep, size_t size)
> +{
> + struct mem_cgroup *memcg;
> +
> + rcu_read_lock();
```

```
> + memcg = cachep->memcg_params.memcg;
> + rcu_read_unlock();
> +
> + /*
> +  * The classical disabled check won't work
> +  * for uncharge, since it is possible that the user enabled
> +  * kmem tracking, allocated, and then disabled.
> +  *
> +  * We trust if there is a memcg associated with the slab,
> +  * it is a valid allocation
> +  */
> + if (!memcg)
> +  return;
> +
> + memcg_uncharge_kmem(memcg, size);
> +}
> +EXPORT_SYMBOL(__mem_cgroup_uncharge_slab);
> +
> +static void memcg_slab_init(struct mem_cgroup *memcg)
> +{
> + int i;
> +
> + for (i = 0; i < MAX_KMEM_CACHE_TYPES; i++)
> +  memcg->slabs[i] = NULL;
> +}
>  #endif /* CONFIG_CGROUP_MEM_RES_CTLR_KMEM */
>
>  static void drain_all_stock_async(struct mem_cgroup *memcg);
> @@ -4760,7 +5071,11 @@ static struct cftype kmem_cgroup_files[] = {
>
>  static int memcg_init_kmem(struct mem_cgroup *memcg, struct cgroup_subsys *ss)
>  {
> - return mem_cgroup_sockets_init(memcg, ss);
> + int ret = mem_cgroup_sockets_init(memcg, ss);
> +
> + if (!ret)
> +  memcg_slab_init(memcg);
> + return ret;
>  };
>
>  static void kmem_cgroup_destroy(struct mem_cgroup *memcg)
> @@ -5777,3 +6092,65 @@ static int __init enable_swap_account(char *s)
>  __setup("swapaccount=", enable_swap_account);
>
>  #endif
> +
> +#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
> +int memcg_charge_kmem(struct mem_cgroup *memcg, gfp_t gfp, s64 delta)
```

```
> +{
> + struct res_counter *fail_res;
> + struct mem_cgroup *_memcg;
> + int may_oom, ret;
> + bool nofail = false;
> +
> + may_oom = (gfp & __GFP_WAIT) && (gfp & __GFP_FS) &&
> +     !(gfp & __GFP_NORETRY);
> +
> + ret = 0;
> +
> + if (!memcg)
> +  return ret;
> +
> + _memcg = memcg;
> + ret = __mem_cgroup_try_charge(NULL, gfp, delta / PAGE_SIZE,
> +     &_memcg, may_oom);
> +
> + if ((ret == -EINTR) || (ret && (gfp & __GFP_NOFAIL)))  {
> +  nofail = true;
> +  /*
> +   * __mem_cgroup_try_charge() chose to bypass to root due
> +   * to OOM kill or fatal signal.
> +   * Since our only options are to either fail the
> +   * allocation or charge it to this cgroup, force the
> +   * change, going above the limit if needed.
> +   */
> +  res_counter_charge_nofail(&memcg->res, delta, &fail_res);
> +  if (do_swap_account)
> +   res_counter_charge_nofail(&memcg->memsw, delta,
> +       &fail_res);
> + } else if (ret == -ENOMEM)
> +  return ret;
> +
> + if (nofail)
> +  res_counter_charge_nofail(&memcg->kmem, delta, &fail_res);
> + else
> +  ret = res_counter_charge(&memcg->kmem, delta, &fail_res);
```

Ouch, you allow usage > limit ? It's BUG.

IMHO, if GFP_NOFAIL, memcg accounting should be skipped. Please

```
if (gfp_mask & __GFP_NOFAIL)
 return 0;
```

Or avoid calling memcg_charge_kmem() you can do that as you do in patch 19/29,

I guess you can use a trick like

```
== in 19/29
+ if (!current->mm || atomic_read(&current->memcg_kmem_skip_account))
+  return cachep;
+
 gfp |=  cachep->allocflags;
==

== change like this
 gfp |= cachep->allocflags;

 if (!current->mm || current->memcg_kmem_skip_account || gfp & __GFP_NOFAIL))
==
```

Is this difficult ?

Thanks,
-Kame