Subject: Re: [PATCH v2 06/11] memcg: kmem controller infrastructure
Posted by Glauber Costa on Wed, 22 Aug 2012 08:35:24 GMT

View Forum Message <> Reply to Message

```
On 08/22/2012 01:50 AM, Greg Thelen wrote:
> On Thu, Aug 09 2012, Glauber Costa wrote:
>
>> This patch introduces infrastructure for tracking kernel memory pages to
>> a given memcg. This will happen whenever the caller includes the flag
>> __GFP_KMEMCG flag, and the task belong to a memcg other than the root.
>>
>> In memcontrol.h those functions are wrapped in inline accessors.  The
>> idea is to later on, patch those with static branches, so we don't incur
>> any overhead when no mem cgroups with limited kmem are being used.
>>
>> [ v2: improved comments and standardized function names ]
>>
>> 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>
>> ---
>>  include/linux/memcontrol.h |  79 +++++++++++++++++++
>>  mm/memcontrol.c            | 185 +++++++++++++++++++++++++++++++++++++++++++++
>>  2 files changed, 264 insertions(+)
>>
>> diff --git a/include/linux/memcontrol.h b/include/linux/memcontrol.h
>> index 8d9489f..75b247e 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;
>> @@ -399,6 +400,11 @@ struct sock;
>>  #ifdef CONFIG_MEMCG_KMEM
>>  void sock_update_memcg(struct sock *sk);
>>  void sock_release_memcg(struct sock *sk);
>> +
>> +#define memcg_kmem_on 1
>> +bool __memcg_kmem_new_page(gfp_t gfp, void *handle, int order);
>> +void __memcg_kmem_commit_page(struct page *page, void *handle, int order);
>> +void __memcg_kmem_free_page(struct page *page, int order);
>>  #else
>>  static inline void sock_update_memcg(struct sock *sk)
>>  {
>> @@ -406,6 +412,79 @@ static inline void sock_update_memcg(struct sock *sk)
>>  static inline void sock_release_memcg(struct sock *sk)
>>  {
>>  }
>> +
>> +#define memcg_kmem_on 0
>> +static inline bool
>> +__memcg_kmem_new_page(gfp_t gfp, void *handle, int order)
>> +{
>> + return false;
>> +}
>> +
>> +static inline void  __memcg_kmem_free_page(struct page *page, int order)
>> +{
>> +}
>> +
>> +static inline void
>> +__memcg_kmem_commit_page(struct page *page, struct mem_cgroup *handle, int order)
>> +{
>> +}
>>  #endif /* CONFIG_MEMCG_KMEM */
>> +
>> +/**
>> + * memcg_kmem_new_page: verify if a new kmem allocation is allowed.
>> + * @gfp: the gfp allocation flags.
>> + * @handle: a pointer to the memcg this was charged against.
>> + * @order: allocation order.
>> + *
>> + * returns true if the memcg where the current task belongs can hold this
>> + * allocation.
>> + *
>> + * We return true automatically if this allocation is not to be accounted to
>> + * any memcg.
>> + */
>> +static __always_inline bool
>> +memcg_kmem_new_page(gfp_t gfp, void *handle, int order)
>> +{
>> + if (!memcg_kmem_on)
>> +  return true;
>> + if (!(gfp & __GFP_KMEMCG) || (gfp & __GFP_NOFAIL))
>> +  return true;
>> + if (in_interrupt() || (!current->mm) || (current->flags & PF_KTHREAD))
>> +  return true;
>> + return __memcg_kmem_new_page(gfp, handle, order);
>> +}
>> +
>> +/**
>> + * memcg_kmem_free_page: uncharge pages from memcg
>> + * @page: pointer to struct page being freed
>> + * @order: allocation order.
>> + *
>> + * there is no need to specify memcg here, since it is embedded in page_cgroup
>> + */
>> +static __always_inline void
>> +memcg_kmem_free_page(struct page *page, int order)
>> +{
>> + if (memcg_kmem_on)
>> +  __memcg_kmem_free_page(page, order);
>> +}
>> +
>> +/**
>> + * memcg_kmem_commit_page: embeds correct memcg in a page
>> + * @handle: a pointer to the memcg this was charged against.
>> + * @page: pointer to struct page recently allocated
>> + * @handle: the memcg structure we charged against
>> + * @order: allocation order.
>> + *
>> + * Needs to be called after memcg_kmem_new_page, regardless of success or
>> + * failure of the allocation. if @page is NULL, this function will revert the
>> + * charges. Otherwise, it will commit the memcg given by @handle to the
>> + * corresponding page_cgroup.
>> + */
>> +static __always_inline void
>> +memcg_kmem_commit_page(struct page *page, struct mem_cgroup *handle, int order)
>> +{
>> + if (memcg_kmem_on)
>> +  __memcg_kmem_commit_page(page, handle, order);
>> +}
>>  #endif /* _LINUX_MEMCONTROL_H */
>>
>> diff --git a/mm/memcontrol.c b/mm/memcontrol.c
>> index 54e93de..e9824c1 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
>> @@ -434,6 +438,9 @@ struct mem_cgroup *mem_cgroup_from_css(struct
cgroup_subsys_state *s)
>>  #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) {
>> @@ -488,6 +495,118 @@ struct cg_proto *tcp_proto_cgroup(struct mem_cgroup *memcg)
>>  }
>>  EXPORT_SYMBOL(tcp_proto_cgroup);
>>  #endif /* CONFIG_INET */
>> +
>> +static inline bool memcg_kmem_enabled(struct mem_cgroup *memcg)
>> +{
>> + return !mem_cgroup_disabled() && !mem_cgroup_is_root(memcg) &&
>> +  memcg->kmem_accounted;
>> +}
>> +
>> +/*
>> + * We need to verify if the allocation against current->mm->owner's memcg is
>> + * possible for the given order. But the page is not allocated yet, so we'll
>> + * need a further commit step to do the final arrangements.
>> + *
>> + * It is possible for the task to switch cgroups in this mean time, so at
>> + * commit time, we can't rely on task conversion any longer.  We'll then use
>> + * the handle argument to return to the caller which cgroup we should commit
>> + * against
>> + *
>> + * Returning true means the allocation is possible.
>> + */
>> +bool __memcg_kmem_new_page(gfp_t gfp, void *_handle, int order)
>> +{
>> + struct mem_cgroup *memcg;
>> + struct mem_cgroup **handle = (struct mem_cgroup **)_handle;
>> + bool ret = true;
>> + size_t size;
>> + struct task_struct *p;
>> +
>> + *handle = NULL;
>> + rcu_read_lock();
>> + p = rcu_dereference(current->mm->owner);
>> + memcg = mem_cgroup_from_task(p);
>> + if (!memcg_kmem_enabled(memcg))
>> +  goto out;
>> +
>> + mem_cgroup_get(memcg);
>> +
>> + size = PAGE_SIZE << order;
>> + ret = memcg_charge_kmem(memcg, gfp, size) == 0;
>> + if (!ret) {
>> +  mem_cgroup_put(memcg);
>> +  goto out;
>> + }
>> +
>> + *handle = memcg;
>> +out:
>> + rcu_read_unlock();
>> + return ret;
>> +}
>> +EXPORT_SYMBOL(__memcg_kmem_new_page);
>> +
>> +void __memcg_kmem_commit_page(struct page *page, void *handle, int order)
>> +{
>> + struct page_cgroup *pc;
>> + struct mem_cgroup *memcg = handle;
>> +
>> + if (!memcg)
>> +  return;
>> +
>> + WARN_ON(mem_cgroup_is_root(memcg));
>> + /* The page allocation must have failed. Revert */
>> + if (!page) {
>> +  size_t size = PAGE_SIZE << order;
>> +
>> +  memcg_uncharge_kmem(memcg, size);
>> +  mem_cgroup_put(memcg);
>> +  return;
>
>> +
>> + pc = lookup_page_cgroup(page);
>> + lock_page_cgroup(pc);
>> + pc->mem_cgroup = memcg;
>> + SetPageCgroupUsed(pc);
>> + unlock_page_cgroup(pc);
>
```

> I have no problem with the code here. But, out of curiosity, why do we
> need to lock the pc here and below in __memcg_kmem_free_page()?
>
> For the allocating side, I don't think that migration or reclaim will be

> manipulating this page. But is there something else that we need the
> locking for?
>
> For the freeing side, it seems that anyone calling
> __memcg_kmem_free_page() is going to be freeing a previously accounted
> page.
>
> I imagine that if we did not need the locking we would still need some
> memory barriers to make sure that modifications to the PG_lru are
> serialized wrt. to kmem modifying PageCgroupUsed here.
>

Unlocking should do that, no?

> Perhaps we're just trying to take a conservative initial implementation
> which is consistent with user visible pages.
>

The way I see it, is not about being conservative, but rather about my
physical safety. It is quite easy and natural to assume that "all
modifications to page cgroup are done under lock". So someone modifying
this later will likely find out about this exception in a rather
unpleasant way. They know where I live, and guns for hire are everywhere.

Note that it is not unreasonable to believe that we can modify this
later. This can be a way out, for example, for the memcg lifecycle problem.

I agree with your analysis and we can ultimately remove it, but if we
cannot pinpoint any performance problems to here, maybe consistency
wins. Also, the locking operation itself is a bit expensive, but the
biggest price is the actual contention. If we'll have nobody contending
for the same page_cgroup, the problem - if exists - shouldn't be that
bad. And if we ever have, the lock is needed.

```
>> +}
>> +
>> +void __memcg_kmem_free_page(struct page *page, int order)
>> +{
>> + 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)) {
>
> When do we expect to find PageCgroupUsed() unset in this routine? Is
> this just to handle the race of someone enabling kmem accounting after
> allocating a page and then later freeing that page?
>

All the time we have a valid memcg. It is marked Used at charge time, so this is how we differentiate between a tracked page and a non-tracked page. Note that even though we explicit mark the freeing call sites with free_allocated_page, etc, not all pc->memcg will be valid. There are unlimited memcgs, bypassed charges, GFP_NOFAIL allocations, etc.