

---

Subject: Re: [PATCH v5 06/14] memcg: kmem controller infrastructure  
Posted by [David Rientjes](#) on Wed, 17 Oct 2012 22:37:32 GMT  
[View Forum Message](#) <> [Reply to Message](#)

---

On Tue, 16 Oct 2012, Glauber Costa wrote:

```
> diff --git a/include/linux/memcontrol.h b/include/linux/memcontrol.h
> index 8d9489f..303a456 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 +400,88 @@ struct sock;
> #ifdef CONFIG_MEMCG_KMEM
> void sock_update_memcg(struct sock *sk);
> void sock_release_memcg(struct sock *sk);
> +
> +static inline bool memcg_kmem_enabled(void)
> +{
> + return true;
> +}
> +
> +bool __memcg_kmem_newpage_charge(gfp_t gfp, struct mem_cgroup **memcg,
> + int order);
> +void __memcg_kmem_commit_charge(struct page *page,
> + struct mem_cgroup *memcg, int order);
> +void __memcg_kmem_uncharge_page(struct page *page, int order);
> +
> +/**
> + * memcg_kmem_newpage_charge: verify if a new kmem allocation is allowed.
> + * @gfp: the gfp allocation flags.
> + * @memcg: 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_newpage_charge(gfp_t gfp, struct mem_cgroup **memcg, int order)
```

```

> +{
> + if (!memcg_kmem_enabled())
> + return true;
> +
> + /*
> + * __GFP_NOFAIL allocations will move on even if charging is not
> + * possible. Therefore we don't even try, and have this allocation
> + * unaccounted. We could in theory charge it with
> + * res_counter_charge_nofail, but we hope those allocations are rare,
> + * and won't be worth the trouble.
> + */
> + if (!(gfp & __GFP_KMEMCG) || (gfp & __GFP_NOFAIL))
> + return true;
> + if (in_interrupt() || (!current->mm) || (current->flags & PF_KTHREAD))
> + return true;
> +
> + /* If the test is dying, just let it go. */
> +     if (unlikely(test_thread_flag(TIF_MEMDIE)
> +         || fatal_signal_pending(current)))
> + return true;

```

This can be simplified to just check fatal\_signal\_pending(), all threads with TIF\_MEMDIE also have a pending SIGKILL.

It also has whitespace damage.

```

> +
> + return __memcg_kmem_newpage_charge(gfp, memcg, order);
> +}
> +
> +/**
> + * memcg_kmem_uncharge_page: uncharge pages from memcg

```

Should be memcg\_kmem\_uncharge\_pages() since it takes an order argument?

```

> + * @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_uncharge_page(struct page *page, int order)
> +{
> + if (memcg_kmem_enabled())
> + __memcg_kmem_uncharge_page(page, order);
> +}
> +
> +/**

```

```

> + * memcg_kmem_commit_charge: embeds correct memcg in a page
> + * @page: pointer to struct page recently allocated
> + * @memcg: the memcg structure we charged against
> + * @order: allocation order.
> + *
> + * Needs to be called after memcg_kmem_newpage_charge, 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 @memcg to the
> + * corresponding page_cgroup.
> + */
> +static __always_inline void
> +memcg_kmem_commit_charge(struct page *page, struct mem_cgroup *memcg, int order)
> +{
> + if (memcg_kmem_enabled() && memcg)
> + __memcg_kmem_commit_charge(page, memcg, order);
> +}
> +
> + #else
> + static inline void sock_update_memcg(struct sock *sk)
> + {
> + @@ -406,6 +489,21 @@ static inline void sock_update_memcg(struct sock *sk)
> + static inline void sock_release_memcg(struct sock *sk)
> + {
> + }
> +
> +
> +static inline bool
> +memcg_kmem_newpage_charge(gfp_t gfp, struct mem_cgroup **memcg, int order)
> +{
> + return true;
> +}
> +
> +static inline void memcg_kmem_uncharge_page(struct page *page, int order)

```

Two spaces.

```

> +{
> +}
> +
> +static inline void
> +memcg_kmem_commit_charge(struct page *page, struct mem_cgroup *memcg, int order)
> +{
> +}
> + #endif /* CONFIG_MEMCG_KMEM */
> + #endif /* _LINUX_MEMCONTROL_H */
>
> diff --git a/mm/memcontrol.c b/mm/memcontrol.c
> index 30eafeb..1182188 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
> @@ -2630,6 +2634,171 @@ static void __mem_cgroup_commit_charge(struct mem_cgroup
*memcg,
> memcg_check_events(memcg, page);
> }
>
> +#ifdef CONFIG_MEMCG_KMEM
> +static inline bool memcg_can_account_kmem(struct mem_cgroup *memcg)
> +{
> + return !mem_cgroup_disabled() && !mem_cgroup_is_root(memcg) &&
> + (memcg->kmem_accounted & KMEM_ACCOUNTED_MASK);
> +}
> +
> +static int memcg_charge_kmem(struct mem_cgroup *memcg, gfp_t gfp, u64 size)
> +{
> + struct res_counter *fail_res;
> + struct mem_cgroup *_memcg;
> + int ret = 0;
> + bool may_oom;
> +
> + ret = res_counter_charge(&memcg->kmem, size, &fail_res);
> + if (ret)
> + return ret;
> +
> + /*
> +  * Conditions under which we can wait for the oom_killer.
> +  * We have to be able to wait, but also, if we can't retry,
> +  * we obviously shouldn't go mess with oom.
> +  */
> + may_oom = (gfp & __GFP_WAIT) && !(gfp & __GFP_NORETRY);

```

What about gfp & \_\_GFP\_FS?

```

> +
> + _memcg = memcg;
> + ret = __mem_cgroup_try_charge(NULL, gfp, size >> PAGE_SHIFT,
> + &_memcg, may_oom);

```

```

> +
> + if (ret == -EINTR) {
> + /*
> + * __mem_cgroup_try_charge() chosed 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, do
> + * it as a temporary condition. But we can't fail. From a
> + * kmem/slab perspective, the cache has already been selected,
> + * by mem_cgroup_get_kmem_cache(), so it is too late to change
> + * our minds. This condition will only trigger if the task
> + * entered memcg_charge_kmem in a sane state, but was
> + * OOM-killed. during __mem_cgroup_try_charge. Tasks that are

```

Looks like some copy-and-paste damage.

```

> + * already dying when the allocation triggers should have been
> + * already directed to the root cgroup.
> + */
> + res_counter_charge_nofail(&memcg->res, size, &fail_res);
> + if (do_swap_account)
> + res_counter_charge_nofail(&memcg->memsw, size,
> + &fail_res);
> + ret = 0;
> + } else if (ret)
> + res_counter_uncharge(&memcg->kmem, size);
> +
> + return ret;
> +}
> +
> +static void memcg_uncharge_kmem(struct mem_cgroup *memcg, u64 size)
> +{
> + res_counter_uncharge(&memcg->kmem, size);
> + res_counter_uncharge(&memcg->res, size);
> + if (do_swap_account)
> + res_counter_uncharge(&memcg->memsw, size);
> +}
> +
> +/*
> + * 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. We could also return the memcg directly and avoid the pointer
> + * passing, but a boolean return value gives better semantics considering
> + * the compiled-out case as well.

```

```

> + *
> + * Returning true means the allocation is possible.
> + */
> + bool
> + __memcg_kmem_newpage_charge(gfp_t gfp, struct mem_cgroup **_memcg, int order)
> + {
> + struct mem_cgroup *memcg;
> + int ret;
> +
> + *_memcg = NULL;
> + memcg = try_get_mem_cgroup_from_mm(current->mm);
> +
> + /*
> + * very rare case described in mem_cgroup_from_task. Unfortunately there
> + * isn't much we can do without complicating this too much, and it would
> + * be gfp-dependent anyway. Just let it go
> + */
> + if (unlikely(!memcg))
> + return true;
> +
> + if (!memcg_can_account_kmem(memcg)) {
> + css_put(&memcg->css);
> + return true;
> + }
> +
> + mem_cgroup_get(memcg);
> +
> + ret = memcg_charge_kmem(memcg, gfp, PAGE_SIZE << order);
> + if (!ret)
> + *_memcg = memcg;
> + else
> + mem_cgroup_put(memcg);
> +
> + css_put(&memcg->css);
> + return (ret == 0);
> + }
> +
> + void __memcg_kmem_commit_charge(struct page *page, struct mem_cgroup *memcg,
> + int order)
> + {
> + struct page_cgroup *pc;
> +
> + VM_BUG_ON(mem_cgroup_is_root(memcg));
> +
> + /* The page allocation failed. Revert */
> + if (!page) {
> + memcg_uncharge_kmem(memcg, PAGE_SIZE << order);
> + mem_cgroup_put(memcg);

```

```

> + return;
> + }
> +
> + pc = lookup_page_cgroup(page);
> + lock_page_cgroup(pc);
> + pc->mem_cgroup = memcg;
> + SetPageCgroupUsed(pc);
> + unlock_page_cgroup(pc);
> +}
> +
> +void __memcg_kmem_uncharge_page(struct page *page, int order)
> +{
> + struct mem_cgroup *memcg = NULL;
> + struct page_cgroup *pc;
> +
> +
> + pc = lookup_page_cgroup(page);
> + /*
> +  * Fast unlocked return. Theoretically might have changed, have to
> +  * check again after locking.
> +  */
> + if (!PageCgroupUsed(pc))
> + return;
> +
> + lock_page_cgroup(pc);
> + if (PageCgroupUsed(pc)) {
> + memcg = pc->mem_cgroup;
> + ClearPageCgroupUsed(pc);
> + }
> + unlock_page_cgroup(pc);
> +
> + /*
> +  * We trust that only if there is a memcg associated with the page, it
> +  * is a valid allocation
> +  */
> + if (!memcg)
> + return;
> +
> + VM_BUG_ON(mem_cgroup_is_root(memcg));
> + memcg_uncharge_kmem(memcg, PAGE_SIZE << order);
> + mem_cgroup_put(memcg);

```

Should this mem\_cgroup\_put() be done conditionally on memcg->kmem\_accounted & KMEM\_ACCOUNTED\_MASK?

The next patch in the series does memcg\_kmem\_newpage\_charge() in the page allocator which will return true for memcg\_can\_account\_kmem() without doing mem\_cgroup\_get().

```
> +}  
> +#endif /* CONFIG_MEMCG_KMEM */  
> +  
> #ifdef CONFIG_TRANSPARENT_HUGEPAGE  
>  
> #define PCGF_NOCOPY_AT_SPLIT (1 << PCG_LOCK | 1 << PCG_MIGRATION)
```

---