Subject: Re: [PATCH v3 06/13] memcg: kmem controller infrastructure
Posted by Michal Hocko on Wed, 26 Sep 2012 15:51:09 GMT
View Forum Message <> Reply to Message

On Tue 18-09-12 18:04:03, 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 acessors. 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.

Could you describe the API a bit here, please? I guess the
kernel user is supposed to call memcg_kmem_newpage_charge and
memcg_kmem_commit_charge resp. memcg_kmem_uncharge_page.
All other kmem functions here are just helpers, right?
>
> [v2: improved comments and standardized function names]
> [v3: handle no longer opaque, functions not exported,
> even more comments]
> [v4: reworked Used bit handling and surroundings for more clarity]
>
> 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 | 97 ++++++++++++++++++++++++++++++
> mm/memcontrol.c | 177 ++
> 2 files changed, 274 insertions(+)
>
> diff --git a/include/linux/memcontrol.h b/include/linux/memcontrol.h
> index 8d9489f..82ede9a 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 +400,17 @@ 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;
> +}
> +
> +extern bool __memcg_kmem_newpage_charge(gfp_t gfp, struct mem_cgroup **memcg,
> +   int order);
> +extern void __memcg_kmem_commit_charge(struct page *page,
> +   struct mem_cgroup *memcg, int order);
> +extern void __memcg_kmem_uncharge_page(struct page *page, int order);
> #else
> static inline void sock_update_memcg(struct sock *sk)
> {
> @@ -406,6 +418,91 @@ static inline void sock_update_memcg(struct sock *sk)
> static inline void sock_release_memcg(struct sock *sk)
> {
> }
> +
> +static inline bool memcg_kmem_enabled(void)
> +{
> + return false;
> +}
> +
> +static inline bool
> +__memcg_kmem_newpage_charge(gfp_t gfp, struct mem_cgroup **memcg, int order)
> +{
> + return false;
> +}
> +
> +static inline void __memcg_kmem_uncharge_page(struct page *page, int order)
> +{
> +}
> +
> +static inline void
> +__memcg_kmem_commit_charge(struct page *page, struct mem_cgroup *memcg, int order)
> +{
> +}

I think we shouldn't care about these for !MEMCG_KMEM. It should be
sufficient to define the main three functions bellow as return true
resp. NOOP. This would reduce the code churn a bit and also make it
better maintainable.

> #endif /* CONFIG_MEMCG_KMEM */
> +

```
> +/**
> + * 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;
> + return __memcg_kmem_newpage_charge(gfp, memcg, order);
> +}
> +
> +/**
> + * memcg_kmem_uncharge_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_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
```

> + * @memcg: a pointer to the memcg this was charged against.
>   ^^^^^^^
remove this one?

> + * @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);
> +}
> #endif /* _LINUX_MEMCONTROL_H */
>
> diff --git a/mm/memcontrol.c b/mm/memcontrol.c
> index f3fd354..0f36a01 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
> @@ -426,6 +430,9 @@ 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, u64 size);
> +static void memcg_uncharge_kmem(struct mem_cgroup *memcg, u64 size);
> +

Why the forward declarations here? We can simply move definitions up
before they are used for the first time, can't we? Besides that they are
never used/defined from outside of KMEM_MEMCG.

> void sock_update_memcg(struct sock *sk)
> {
>   if (mem_cgroup_sockets_enabled) {
> @@ -480,6 +487,110 @@ struct cg_proto *tcp_proto_cgroup(struct mem_cgroup *memcg)
> }
> EXPORT_SYMBOL(tcp_proto_cgroup);
> #endif /* CONFIG_INET */
> +
> +static inline bool memcg_can_account_kmem(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. 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;
> + bool ret;
> + struct task_struct *p;

Johannes likes christmas trees ;) and /me would like to remove `p' and
use mem_cgroup_from_task(rcu_dereference(current->mm->owner)) same as we
do at other places (I guess it will be checkpatch safe).

> +
> + *_memcg = NULL;
> + rcu_read_lock();
> + p = rcu_dereference(current->mm->owner);
> + memcg = mem_cgroup_from_task(p);

mem_cgroup_from_task says it can return NULL. Do we care here? If not
then please put VM_BUG_ON(!memcg) here.

> + rcu_read_unlock();
> +
> + if (!memcg_can_account_kmem(memcg))
> + return true;
> +
> + mem_cgroup_get(memcg);

I am confused. Why do we take a reference to memcg rather than css_get
here? Ahh it is because we keep the reference while the page is
allocated, right? Comment please.

I am still not sure whether we need css_get here as well. How do you
know that the current is not moved in parallel and it is a last task in
a group which then can go away?

> +
> + ret = memcg_charge_kmem(memcg, gfp, PAGE_SIZE << order) == 0;
> + if (ret)
> + *_memcg = memcg;
> + else
> + mem_cgroup_put(memcg);
> +
> + return ret;
> +}
> +
> +void __memcg_kmem_commit_charge(struct page *page, struct mem_cgroup *memcg,
> +   int order)
> +{
> + struct page_cgroup *pc;
> +
> + WARN_ON(mem_cgroup_is_root(memcg));

Why the warn? Nobody should use this directly and
memcg_kmem_commit_charge takes care of the root same as
__memcg_kmem_newpage_charge does. If it is for correctness then it
should be VM_BUG_ON.

> +
> + /* The page allocation failed. Revert */
> + if (!page) {
> + memcg_uncharge_kmem(memcg, PAGE_SIZE << order);
> + 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);
> +
> + /*
> + * Checking if kmem accounted is enabled won't work for uncharge, since
> + * it is possible that the user enabled kmem tracking, allocated, and
> + * then disabled it again.

disabling cannot happen, right?

> + *
> + * 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));

Same as above I do not see a reason for warn here. It just adds a code
and if you want it for debugging then VM_BUG_ON sounds more appropriate.
/me thinks

> + memcg_uncharge_kmem(memcg, PAGE_SIZE << order);
> + mem_cgroup_put(memcg);
> +}

> #endif /* CONFIG_MEMCG_KMEM */
>
> #if defined(CONFIG_INET) && defined(CONFIG_MEMCG_KMEM)
> @@ -5700,3 +5811,69 @@ static int __init enable_swap_account(char *s)
> __setup("swapaccount=", enable_swap_account);
>
> #endif
> +
> +#ifdef CONFIG_MEMCG_KMEM
> +int memcg_charge_kmem(struct mem_cgroup *memcg, gfp_t gfp, u64 size)
> +{
> + struct res_counter *fail_res;
> + struct mem_cgroup *_memcg;
> + int ret;
> + bool may_oom;
> + bool nofail = false;
> +
> + may_oom = (gfp & __GFP_WAIT) && (gfp & __GFP_FS) &&
> +   !(gfp & __GFP_NORETRY);

A comment please? Why __GFP_IO is not considered for example?

> +
> + ret = 0;
> +
> + if (!memcg)
> + return ret;

How can we get a NULL memcg here without blowing in
__memcg_kmem_newpage_charge?

> +
> + _memcg = memcg;
> + ret = __mem_cgroup_try_charge(NULL, gfp, size / PAGE_SIZE,

me likes >> PAGE_SHIFT more.

> +   &_memcg, may_oom);
> +
> + if (ret == -EINTR) {
> + nofail = true;
> + /*
> +   * __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
> +   */
> +  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 == -ENOMEM)
> +  return ret;
> +
> + if (nofail)
> +  res_counter_charge_nofail(&memcg->kmem, size, &fail_res);
> + else
> +  ret = res_counter_charge(&memcg->kmem, size, &fail_res);
> +
> + if (ret) {
> +  res_counter_uncharge(&memcg->res, size);
> +  if (do_swap_account)
> +   res_counter_uncharge(&memcg->memsw, size);
> + }
```

You could save few lines and get rid of the strange nofail by:

```
[...]
+  res_counter_charge_nofail(&memcg->res, size, &fail_res);
+  if (do_swap_account)
+   res_counter_charge_nofail(&memcg->memsw, size,
+         &fail_res);
+  res_counter_charge_nofail(&memcg->kmem, size, &fail_res);
+  return 0;
+ } else if (ret == -ENOMEM)
+  return ret;
+ else
+  ret = res_counter_charge(&memcg->kmem, size, &fail_res);
+
+ if (ret) {
+  res_counter_uncharge(&memcg->res, size);
+  if (do_swap_account)
+   res_counter_uncharge(&memcg->memsw, size);
+ }
```

```
> +
> + return ret;
> +}
> +
> +void memcg_uncharge_kmem(struct mem_cgroup *memcg, u64 size)
> +{
> + if (!memcg)
> +  return;
```

> +
> + res_counter_uncharge(&memcg->kmem, size);
> + res_counter_uncharge(&memcg->res, size);
> + if (do_swap_account)
> + res_counter_uncharge(&memcg->memsw, size);
> +}
> +#endif /* CONFIG_MEMCG_KMEM */
> --
> 1.7.11.4
>
> --
> To unsubscribe from this list: send the line "unsubscribe cgroups" in
> the body of a message to majordomo@vger.kernel.org
> More majordomo info at http://vger.kernel.org/majordomo-info.html

--
Michal Hocko
SUSE Labs
