Subject: Re: [PATCH 06/11] memcg: kmem controller infrastructure
Posted by akpm on Mon, 25 Jun 2012 23:17:20 GMT
View Forum Message <> Reply to Message

On Mon, 25 Jun 2012 18:15:23 +0400
Glauber Costa <glommer@parallels.com> 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 jump labels, so we don't
> incur any overhead when no mem cgroups are being used.
>
>
> ...
>
> @@ -416,6 +423,43 @@ static inline void sock_update_memcg(struct sock *sk)
> static inline void sock_release_memcg(struct sock *sk)
> {
> }
> +
> +#define mem_cgroup_kmem_on 0
> +#define __mem_cgroup_new_kmem_page(a, b, c) false
> +#define __mem_cgroup_free_kmem_page(a,b)
> +#define __mem_cgroup_commit_kmem_page(a, b, c)

I suggest that the naming consistently follow the model
"mem_cgroup_kmem_foo". So "mem_cgroup_kmem_" becomes the well-known
identifier for this subsystem.

Then, s/mem_cgroup/memcg/g/ - show us some mercy here!

> +#define is_kmem_tracked_alloc (false)

memcg_kmem_tracked_alloc, perhaps. But what does this actually do?

<looks>

eww, ick.

> +#define is_kmem_tracked_alloc (gfp & __GFP_KMEMCG)

What Tejun said. This:

/*

```
 * Nice comment goes here
 */
static inline bool memcg_kmem_tracked_alloc(gfp_t gfp)
{
 return gfp & __GFP_KMEMCG;
}
```

```
>  #endif /* CONFIG_CGROUP_MEM_RES_CTLR_KMEM */
> +
> +static __always_inline
> +bool mem_cgroup_new_kmem_page(gfp_t gfp, void *handle, int order)
```

memcg_kmem_new_page().

```
> +{
> + if (!mem_cgroup_kmem_on)
> +  return true;
> + if (!is_kmem_tracked_alloc)
> +  return true;
> + if (!current->mm)
> +  return true;
> + if (in_interrupt())
> +  return true;
> + if (gfp & __GFP_NOFAIL)
> +  return true;
> + return __mem_cgroup_new_kmem_page(gfp, handle, order);
> +}
```

Add documentation, please. The semantics of the return value are
inscrutable.

```
> +static __always_inline
> +void mem_cgroup_free_kmem_page(struct page *page, int order)
> +{
> + if (mem_cgroup_kmem_on)
> +  __mem_cgroup_free_kmem_page(page, order);
> +}
```

memcg_kmem_free_page().

```
> +static __always_inline
> +void mem_cgroup_commit_kmem_page(struct page *page, struct mem_cgroup *handle,
> +     int order)
> +{
> + if (mem_cgroup_kmem_on)
> +  __mem_cgroup_commit_kmem_page(page, handle, order);
> +}
```

memcg_kmem_commit_page().

> #endif /* _LINUX_MEMCONTROL_H */
>
>
> ...
>
> +static inline bool mem_cgroup_kmem_enabled(struct mem_cgroup *memcg)
> +{
> + return !mem_cgroup_disabled() && memcg &&
> + !mem_cgroup_is_root(memcg) && memcg->kmem_accounted;
> +}

Does this really need to handle a memcg==NULL?

> +bool __mem_cgroup_new_kmem_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 (!mem_cgroup_kmem_enabled(memcg))
> + goto out;
> +
> + mem_cgroup_get(memcg);
> +
> + size = (1 << order) << PAGE_SHIFT;

size = PAGE_SIZE << order;

is simpler and more typical.

> + ret = memcg_charge_kmem(memcg, gfp, size) == 0;

Odd. memcg_charge_kmem() returns a nice errno, buit this conversion
just drops that information on the floor. If the
mem_cgroup_new_kmem_page() return value had been documented, I might
have been able to understand the thinking here. But it wasn't, so I
couldn't.

> + if (!ret) {
> + mem_cgroup_put(memcg);

> + goto out;
> + }
> +
> + *handle = memcg;
> +out:
> + rcu_read_unlock();
> + return ret;
> +}
> +EXPORT_SYMBOL(__mem_cgroup_new_kmem_page);
> +
> +void __mem_cgroup_commit_kmem_page(struct page *page, void *handle, int order)
> +{
> + struct page_cgroup *pc;
> + struct mem_cgroup *memcg = handle;
> + size_t size;
> +
> + if (!memcg)
> + return;
> +
> + WARN_ON(mem_cgroup_is_root(memcg));
> + /* The page allocation must have failed. Revert */
> + if (!page) {
> + size = (1 << order) << PAGE_SHIFT;

 PAGE_SIZE << order

> + memcg_uncharge_kmem(memcg, size);
> + mem_cgroup_put(memcg);
> + return;
> + }

This all looks a bit odd. After a failure you run a commit which
undoes the speculative charging. I guess it makes sense. It's the
sort of thing which can be expounded upon in the documentation whcih
isn't there, sigh.

> + pc = lookup_page_cgroup(page);
> + lock_page_cgroup(pc);
> + pc->mem_cgroup = memcg;
> + SetPageCgroupUsed(pc);
> + unlock_page_cgroup(pc);
> +}

missed a newline there.

> +void __mem_cgroup_free_kmem_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)) {
> + unlock_page_cgroup(pc);
> + return;
> + }
> + ClearPageCgroupUsed(pc);
> + unlock_page_cgroup(pc);
> +
> + /*
> + * The classical disabled check won't work

What is "The classical disabled check"? Be specific. Testing
mem_cgroup_kmem_on?

> + * 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 << order) << PAGE_SHIFT;

PAGE_SIZE << order

> + memcg_uncharge_kmem(memcg, size);
> + mem_cgroup_put(memcg);
> +}
> +EXPORT_SYMBOL(__mem_cgroup_free_kmem_page);
> #endif /* CONFIG_CGROUP_MEM_RES_CTLR_KMEM */
>
> #if defined(CONFIG_INET) && defined(CONFIG_CGROUP_MEM_RES_CTLR_KMEM)
> @@ -5645,3 +5751,69 @@ static int __init enable_swap_account(char *s)
> __setup("swapaccount=", enable_swap_account);
>
> #endif

> +
> +#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM

gargh. CONFIG_MEMCG_KMEM, please!

> +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);

may_oom should have bool type.

> + ret = 0;
> +
> + if (!memcg)
> + return ret;
> +
> + _memcg = memcg;
> + ret = __mem_cgroup_try_charge(NULL, gfp, delta / PAGE_SIZE,
> +   &_memcg, may_oom);
> +
> + if (ret == -EINTR) {
> + nofail = true;
> + /*
> +   * __mem_cgroup_try_charge() chose to bypass to root due
> +   * to OOM kill or fatal signal.

Is "bypass" correct? Maybe "fall back"?

> +   * 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, delta, &fail_res);
> + if (do_swap_account)
> +   res_counter_charge_nofail(&memcg->memsw, delta,
> +   &fail_res);
> + ret = 0;
> + } 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);
> +
> + if (ret) {
> +  res_counter_uncharge(&memcg->res, delta);
> +  if (do_swap_account)
> +   res_counter_uncharge(&memcg->memsw, delta);
> + }
> +
> + return ret;
> +}
>
> ...
>
```