Subject: Re: [PATCH v3 06/13] memcg: kmem controller infrastructure
Posted by Michal Hocko on Thu, 27 Sep 2012 13:44:32 GMT
View Forum Message <> Reply to Message

On Thu 27-09-12 15:31:57, Glauber Costa wrote:
> On 09/26/2012 07:51 PM, Michal Hocko wrote:
> > On Tue 18-09-12 18:04:03, Glauber Costa wrote:
[...]
> >> + *_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.
> ok.
>
> >
> > 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?
>
> the reference count aquired by mem_cgroup_get will still prevent the
> memcg from going away, no?

Yes but you are outside of the rcu now and we usually do css_get before
we rcu_unlock. mem_cgroup_get just makes sure the group doesn't get
deallocated but it could be gone before you call it. Or I am just
confused - these 2 levels of ref counting is really not nice.

Anyway, I have just noticed that __mem_cgroup_try_charge does
VM_BUG_ON(css_is_removed(&memcg->css)) on a given memcg so you should
keep css ref count up as well.

> >> + /* 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?
> >
> not anymore, right. I can update the comment,
```

yes, it is confusing

> but I still believe it is a lot saner to trust information in
> page_cgroup.

I have no objections against that. PageCgroupUsed test and using
pc->mem_cgroup is fine.

```
> >> +#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?
> >
> >
>
> Actually, I believe testing for GFP_WAIT and !GFP_NORETRY would be enough.
>
> The rationale here is, of course, under which circumstance would it be
> valid to call the oom killer? Which is, if the allocation can wait, and
> can retry.
```

Yes __GFP_WAIT is clear because memcg OOM can wait for arbitrary amount
of time (wait for userspace action on oom_control). __GFP_NORETRY
couldn't get to oom before because oom was excluded explicitely for THP
and migration didn't go through the charging path to reach the oom.
But I do agree that __GFP_NORETRY allocations shouldn't cause the OOM
because we should rather fail the allocation from kernel rather than
shoot something.

--
Michal Hocko
SUSE Labs