Subject: Re: [PATCH v2 06/11] memcg: kmem controller infrastructure
Posted by Glauber Costa on Mon, 13 Aug 2012 08:28:14 GMT
View Forum Message <> Reply to Message

>> > + * 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);
>> > +}
> Doesn't this 2 functions has no short-cuts ?

Sorry kame, what exactly do you mean?

> if (memcg_kmem_on && handle) ?
I guess this can be done to avoid a function call.

> Maybe free() needs to access page_cgroup...
>
Can you also be a bit more specific here?

>> > +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);
>> > +
> This mem_cgroup_get() will be a potentioal performance problem.
> Don't you have good idea to avoid accessing atomic counter here ?
> I think some kind of percpu counter or a feature to disable "move task"
> will be a help.

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

> shouldn't this happen after checking "Used" bit ?
> Ah, BTW, why do you need to clear pc->memcg ?

As for clearing pc->memcg, I think I'm just being overzealous. I can't
foresee any problems due to removing it.

As for the Used bit, what difference does it make when we clear it?

```
>> > + if (!PageCgroupUsed(pc)) {
>> > +  unlock_page_cgroup(pc);
>> > +  return;
>> > + }
>> > + 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.
>> > +  *
>> > +  * 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;
>> > + memcg_uncharge_kmem(memcg, size);
>> > + mem_cgroup_put(memcg);
> Why do we need ref-counting here ? kmem res_counter cannot work as
> reference ?
This is of course the pair of the mem_cgroup_get() you commented on
earlier. If we need one, we need the other. If we don't need one, we
don't need the other =)

The guarantee we're trying to give here is that the memcg structure will
stay around while there are dangling charges to kmem, that we decided
not to move (remember: moving it for the stack is simple, for the slab
is very complicated and ill-defined, and I believe it is better to treat
all kmem equally here)

So maybe we can be clever here, and avoid reference counting at all
times. We call mem_cgroup_get() when the first charge occurs, and then
go for mem_cgroup_put() when our count reaches 0.

What do you think about that?

>> > +#ifdef CONFIG_MEMCG_KMEM
>> > +int memcg_charge_kmem(struct mem_cgroup *memcg, gfp_t gfp, s64 delta)
>> > +{
> What does 'delta' means ?
>
I can change it to something like nr_bytes, more informative.

>> > + 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);
>> > +
>> > + 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() 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, delta, &fail_res);
>> > +  if (do_swap_account)
>> > +   res_counter_charge_nofail(&memcg->memsw, delta,
>> > +         &fail_res);
>> > +  ret = 0;
> Hm, you returns 0 and this charge may never be uncharged....right ?
>
```

Can't see why. By returning 0 we inform our caller that the allocation succeeded. It is up to him to undo it later through a call to uncharge.
