

---

Subject: Re: [PATCH v2 02/13] memcg: Kernel memory accounting infrastructure.  
Posted by [Glauber Costa](#) on Sun, 11 Mar 2012 08:12:04 GMT

[View Forum Message](#) <> [Reply to Message](#)

---

On 03/10/2012 12:39 AM, Suleiman Souhlal wrote:

> Enabled with CONFIG\_CGROUP\_MEM\_RES\_CTLR\_KMEM.

>

> Adds the following files:

> - memory.kmem.independent\_kmem\_limit

> - memory.kmem.usage\_in\_bytes

> - memory.kmem.limit\_in\_bytes

>

> Signed-off-by: Suleiman Souhlal<suleiman@google.com>

> ---

> mm/memcontrol.c | 136

+++++

> 1 files changed, 135 insertions(+), 1 deletions(-)

>

> diff --git a/mm/memcontrol.c b/mm/memcontrol.c

> index 37ad2cb..e6fd558 100644

> --- a/mm/memcontrol.c

> +++ b/mm/memcontrol.c

> @@ -220,6 +220,10 @@ enum memcg\_flags {

> \*/

> MEMCG\_MEMSW\_IS\_MINIMUM, /\* Set when res.limit == memsw.limit \*/

> MEMCG\_OOM\_KILL\_DISABLE, /\* OOM-Killer disable \*/

> + MEMCG\_INDEPENDENT\_KMEM\_LIMIT, /\*

> + \* kernel memory is not counted in

> + \* memory.usage\_in\_bytes

> + \*/

> };

>

> /\*

> @@ -244,6 +248,10 @@ struct mem\_cgroup {

> \*/

> struct res\_counter memsw;

> /\*

> + \* the counter to account for kernel memory usage.

> + \*/

> + struct res\_counter kmem;

> + /\*

> \* Per cgroup active and inactive list, similar to the

> \* per zone LRU lists.

> \*/

> @@ -355,6 +363,7 @@ enum charge\_type {

> #define \_MEM (0)

> #define \_MEMSWAP (1)

> #define \_OOM\_TYPE (2)

```

> +#define _KMEM (3)
> #define MEMFILE_PRIVATE(x, val) (((x)<< 16) | (val))
> #define MEMFILE_TYPE(val) (((val)>> 16)& 0xffff)
> #define MEMFILE_ATTR(val) ((val)& 0xffff)
> @@ -371,6 +380,8 @@ enum charge_type {
>
> static void mem_cgroup_get(struct mem_cgroup *memcg);
> static void mem_cgroup_put(struct mem_cgroup *memcg);
> +static void memcg_kmem_init(struct mem_cgroup *memcg,
> + struct mem_cgroup *parent);
>
> static inline bool
> mem_cgroup_test_flag(const struct mem_cgroup *memcg, enum memcg_flags flag)
> @@ -1435,6 +1446,10 @@ done:
> res_counter_read_u64(&memcg->memsw, RES_USAGE)>> 10,
> res_counter_read_u64(&memcg->memsw, RES_LIMIT)>> 10,
> res_counter_read_u64(&memcg->memsw, RES_FAILCNT));
> + printk(KERN_INFO "kmem: usage %lluKB, limit %lluKB, failcnt %llu\n",
> + res_counter_read_u64(&memcg->kmem, RES_USAGE)>> 10,
> + res_counter_read_u64(&memcg->kmem, RES_LIMIT)>> 10,
> + res_counter_read_u64(&memcg->kmem, RES_FAILCNT));
> }
>
> /*
> @@ -3868,6 +3883,9 @@ static u64 mem_cgroup_read(struct cgroup *cont, struct cftype *cft)
> else
> val = res_counter_read_u64(&memcg->memsw, name);
> break;
> + case _KMEM:
> + val = res_counter_read_u64(&memcg->kmem, name);
> + break;
> default:
> BUG();
> break;
> @@ -3900,8 +3918,15 @@ static int mem_cgroup_write(struct cgroup *cont, struct cftype *cft,
> break;
> if (type == _MEM)
> ret = mem_cgroup_resize_limit(memcg, val);
> - else
> + else if (type == _MEMSWAP)
> ret = mem_cgroup_resize_memsw_limit(memcg, val);
> + else if (type == _KMEM) {
> + if (!mem_cgroup_test_flag(memcg,
> + MEMCG_INDEPENDENT_KMEM_LIMIT))
> + return -EINVAL;
> + ret = res_counter_set_limit(&memcg->kmem, val);
> + } else
> + return -EINVAL;

```

```

> break;
> case RES_SOFT_LIMIT:
> ret = res_counter_memparse_write_strategy(buffer,&val);
> @@ -4606,8 +4631,56 @@ static int mem_control_numa_stat_open(struct inode *unused,
struct file *file)
> #endif /* CONFIG_NUMA */
>
> #ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
> +static u64
> +mem_cgroup_independent_kmem_limit_read(struct cgroup *cgrp, struct cftype *cft)
> +{
> + return mem_cgroup_test_flag(mem_cgroup_from_cont(cgrp),
> + MEMCG_INDEPENDENT_KMEM_LIMIT);
> +}
> +
> +static int mem_cgroup_independent_kmem_limit_write(struct cgroup *cgrp,
> + struct cftype *cft, u64 val)
> +{
> + struct mem_cgroup *memcg;
> +
> + memcg = mem_cgroup_from_cont(cgrp);
> + if (val)
> + mem_cgroup_set_flag(memcg, MEMCG_INDEPENDENT_KMEM_LIMIT);
> + else {
> + mem_cgroup_clear_flag(memcg, MEMCG_INDEPENDENT_KMEM_LIMIT);
> + res_counter_set_limit(&memcg->kmem, RESOURCE_MAX);
> + }
> +
> + return 0;
> +}

```

We need this test to be a bit more strict.

This is what I have in my current version:

```

struct mem_cgroup *memcg = mem_cgroup_from_cont(cgroup);
struct mem_cgroup *parent = parent_mem_cgroup(memcg);

```

```

val = !!val;

```

```

if (!parent || !parent->use_hierarchy || mem_cgroup_is_root(parent)) {
    if (list_empty(&cgroup->children))
        memcg->kmem_independent_accounting = val;
    else
        return -EBUSY;
} else
    return -EINVAL;

```

```

return 0;

```

Over yours, it basically:

- \* Makes sure this has no effect on root cgroup
- \* disallow changing that when we already have children.

Also, I have a TODO item on that: We need to make sure that no memory was already accounted. It was a bit tricky for me, because I was not charging kmem when !independent. But since you are always charging kmem, and you are so far facing no opposition on that particular point, maybe you can also test if RES\_USAGE == 0 for memcg->kmem.

```
> +static struct cftype kmem_cgroup_files[] = {
> + {
> + .name = "kmem.independent_kmem_limit",
> + .write_u64 = mem_cgroup_independent_kmem_limit_write,
> + .read_u64 = mem_cgroup_independent_kmem_limit_read,
> + },
> + {
> + .name = "kmem.limit_in_bytes",
> + .private = MEMFILE_PRIVATE(_KMEM, RES_LIMIT),
> + .write_string = mem_cgroup_write,
> + .read_u64 = mem_cgroup_read,
> + },
> + {
> + .name = "kmem.usage_in_bytes",
> + .private = MEMFILE_PRIVATE(_KMEM, RES_USAGE),
> + .read_u64 = mem_cgroup_read,
> + },
> +};
> +
> static int register_kmem_files(struct cgroup *cont, struct cgroup_subsys *ss)
> {
> + int ret;
> +
> + ret = cgroup_add_files(cont, ss, kmem_cgroup_files,
> + ARRAY_SIZE(kmem_cgroup_files));
> + if (ret)
> + return ret;
> /*
> * Part of this would be better living in a separate allocation
> * function, leaving us with just the cgroup tree population work.
> @@ -4621,6 +4694,10 @@ static int register_kmem_files(struct cgroup *cont, struct
cgroup_subsys *ss)
> static void kmem_cgroup_destroy(struct cgroup_subsys *ss,
> struct cgroup *cont)
> {
> + struct mem_cgroup *memcg;
> +
```

```

> + memcg = mem_cgroup_from_cont(cont);
> + BUG_ON(res_counter_read_u64(&memcg->kmem, RES_USAGE) != 0);
> mem_cgroup_sockets_destroy(cont, ss);
> }
> #else
> @@ -4980,6 +5057,8 @@ mem_cgroup_create(struct cgroup_subsys *ss, struct cgroup *cont)
> }
> memcg->last_scanned_node = MAX_NUMNODES;
> INIT_LIST_HEAD(&memcg->oom_notify);
> + memcg_kmem_init(memcg, parent&& mem_cgroup_test_flag(parent,
> + MEMCG_USE_HIERARCHY) ? parent : NULL);
>
> if (parent)
> memcg->swappiness = mem_cgroup_swappiness(parent);
> @@ -5561,3 +5640,58 @@ static int __init enable_swap_account(char *s)
> __setup("swapaccount=", enable_swap_account);
>
> #endif
> +
> +#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
> +int
> +memcg_charge_kmem(struct mem_cgroup *memcg, gfp_t gfp, long long delta)
> +{
> + struct res_counter *fail_res;
> + struct mem_cgroup *_memcg;
> + int may_oom, ret;
> +
> + may_oom = (gfp & __GFP_WAIT)&& (gfp & __GFP_FS)&&
> + !(gfp & __GFP_NORETRY);
> +
> + ret = 0;
> +
> + _memcg = memcg;
> + if (memcg&& !mem_cgroup_test_flag(memcg,
> + MEMCG_INDEPENDENT_KMEM_LIMIT)) {
> + ret = __mem_cgroup_try_charge(NULL, gfp, delta / PAGE_SIZE,
> + &_amp;_memcg, may_oom);
> + if (ret == -ENOMEM)
> + return ret;
> + }
> +
> + if (memcg&& _memcg == memcg)
> + ret = res_counter_charge(&memcg->kmem, delta, &fail_res);
> +
I don't really follow this if (memcg

```

If you are planning to call this unconditionally from the slab code, I think we should at least exit early if !memcg.

Like this:

```
int may_oom, ret
```

```
if (mem_cgroup_disabled() || !memcg)
    return 0;
```

(And you may need the mem\_cgroup\_disabled() in your code anyway)

```
> + return ret;
> +}
> +
> +void
> +memcg_uncharge_kmem(struct mem_cgroup *memcg, long long delta)
> +{
> + if (memcg)
> + res_counter_uncharge(&memcg->kmem, delta);
> +
> + if (memcg&& !mem_cgroup_test_flag(memcg, MEMCG_INDEPENDENT_KMEM_LIMIT))
> + res_counter_uncharge(&memcg->res, delta);
> +}
mem_cgroup_disabled() here too ?
```

(Actually, I just grep'd and noticed you wrap some code around it in patch 7. It'd make more sense not to call this function when memcg == NULL then ?

mem\_cgroup\_disabled() goes on that function, before the rcu lock.

```
> +
> +static void
> +memcg_kmem_init(struct mem_cgroup *memcg, struct mem_cgroup *parent)
> +{
> + struct res_counter *parent_res;
> +
> + parent_res = NULL;
> + if (parent&& parent != root_mem_cgroup)
> + parent_res = &parent->kmem;
> + res_counter_init(&memcg->kmem, parent_res);
> +}
> +#else /* CONFIG_CGROUP_MEM_RES_CTLR_KMEM */
> +static void
> +memcg_kmem_init(struct mem_cgroup *memcg, struct mem_cgroup *parent)
> +{
> +}
> +#endif /* CONFIG_CGROUP_MEM_RES_CTLR_KMEM */
```

---

---

Subject: Re: [PATCH v2 02/13] memcg: Kernel memory accounting infrastructure.  
Posted by [KAMEZAWA Hiroyuki](#) on Tue, 13 Mar 2012 06:24:46 GMT  
[View Forum Message](#) <> [Reply to Message](#)

---

On Sun, 11 Mar 2012 12:12:04 +0400  
Glauber Costa <glommer@parallels.com> wrote:

```
> On 03/10/2012 12:39 AM, Suleiman Souhlal wrote:
>> Enabled with CONFIG_CGROUP_MEM_RES_CTLR_KMEM.
>>
>> Adds the following files:
>>   - memory.kmem.independent_kmem_limit
>>   - memory.kmem.usage_in_bytes
>>   - memory.kmem.limit_in_bytes
>>
>> Signed-off-by: Suleiman Souhlal<suleiman@google.com>
>> ---
>> mm/memcontrol.c | 136
+++++
>> 1 files changed, 135 insertions(+), 1 deletions(-)
>>
>> diff --git a/mm/memcontrol.c b/mm/memcontrol.c
>> index 37ad2cb..e6fd558 100644
>> --- a/mm/memcontrol.c
>> +++ b/mm/memcontrol.c
>> @@ -220,6 +220,10 @@ enum memcg_flags {
>>   */
>>   MEMCG_MEMSW_IS_MINIMUM, /* Set when res.limit == memsw.limit */
>>   MEMCG_OOM_KILL_DISABLE, /* OOM-Killer disable */
>> + MEMCG_INDEPENDENT_KMEM_LIMIT, /*
>> +   * kernel memory is not counted in
>> +   * memory.usage_in_bytes
>> +   */
>> };
```

After looking codes, I think we need to think whether independent\_kmem\_limit is good or not....

How about adding MEMCG\_KMEM\_ACCOUNT flag instead of this and use only memcg->res/memcg->memsw rather than adding a new counter, memcg->kmem ?

if MEMCG\_KMEM\_ACCOUNT is set -> slab is accounted to mem->res/memsw.  
if MEMCG\_KMEM\_ACCOUNT is not set -> slab is never accounted.

(I think On/Off switch is required..)

Thanks,  
-Kame

---

---

Subject: Re: [PATCH v2 02/13] memcg: Kernel memory accounting infrastructure.  
Posted by [Glauber Costa](#) on Tue, 13 Mar 2012 10:37:30 GMT  
[View Forum Message](#) <> [Reply to Message](#)

---

> After looking codes, I think we need to think  
> whether independent\_kmem\_limit is good or not....  
>  
> How about adding MEMCG\_KMEM\_ACCOUNT flag instead of this and use only  
> memcg->res/memcg->memsw rather than adding a new counter, memcg->kmem ?  
>  
> if MEMCG\_KMEM\_ACCOUNT is set -> slab is accounted to mem->res/memsw.  
> if MEMCG\_KMEM\_ACCOUNT is not set -> slab is never accounted.  
>  
> (I think On/Off switch is required..)  
>  
> Thanks,  
> -Kame  
>

This has been discussed before, I can probably find it in the archives if you want to go back and see it.

But in a nutshell:

1) Supposing independent knob disappear (I will explain in item 2 why I don't want it to), I don't think a flag makes sense either. \*If\* we are planning to enable/disable this, it might make more sense to put some work on it, and allow particular slabs to be enabled/disabled by writing to memory.kmem.slabinfo (-\* would disable all, +\* enable all, +kmallocc\* enable all kmallocc, etc).

Alternatively, what we could do instead, is something similar to what ended up being done for tcp, by request of the network people: if you never touch the limit file, don't bother with it at all, and simply does not account. With Suleiman's lazy allocation infrastructure, that should actually be trivial. And then again, a flag is not necessary, because writing to the limit file does the job, and also convey the meaning well enough.

2) For the kernel itself, we are mostly concerned that a malicious container may pin into memory big amounts of kernel memory which is, ultimately, unreclaimable. In particular, with overcommit allowed scenarios, you can fill the whole physical memory (or at least a significant part) with those objects, well beyond your softlimit allowance, making the creation of further containers impossible. With user memory, you can reclaim the cgroup back to its place. With kernel memory, you can't.

In the particular example of 32-bit boxes, you can easily fill up a



large part of the available 1gb kernel memory with pinned memory and render the whole system unresponsive.

Never allowing the kernel memory to go beyond the soft limit was one of the proposed alternatives. However, it may force you to establish a soft limit where one was not previously needed. Or, establish a low soft limit when you really need a bigger one.

All that said, while reading your message, thinking a bit, the following crossed my mind:

- We can account the slabs to memcg->res normally, and just store the information that this is kernel memory into a percpu counter, as I proposed recently.
- The knob goes away, and becomes implicit: if you ever write anything to memory.kmem.limit\_in\_bytes, we transfer that memory to a separate kmem res\_counter, and proceed from there. We can keep accounting to memcg->res anyway, just that kernel memory will now have a separate limit.
- With this scheme, it may not be necessary to ever have a file memory.kmem.soft\_limit\_in\_bytes. Reclaim is always part of the normal memcg reclaim.

The outlined above would work for us, and make the whole scheme simpler, I believe.

What do you think ?

---

Subject: Re: [PATCH v2 02/13] memcg: Kernel memory accounting infrastructure.  
Posted by [Greg Thelen](#) on Tue, 13 Mar 2012 17:00:58 GMT  
[View Forum Message](#) <> [Reply to Message](#)

Glauber Costa <glommer@parallels.com> writes:

- > 2) For the kernel itself, we are mostly concerned that a malicious container may
- > pin into memory big amounts of kernel memory which is, ultimately,
- > unreclaimable. In particular, with overcommit allowed scenarios, you can fill
- > the whole physical memory (or at least a significant part) with those objects,
- > well beyond your softlimit allowance, making the creation of further containers
- > impossible.
- > With user memory, you can reclaim the cgroup back to its place. With kernel
- > memory, you can't.

In overcommit situations the page allocator starts failing even though memcg page can charge pages. When page allocations fail the oom killer plays a role. Page allocations can fail even without malicious usage of kernel memory (e.g. lots of mlock or anon without swap can fill a machine). I assume that the kernel memory pinned the malicious

containers will be freed or at least become reclaimable once the processes in malicious containers are killed (oom or otherwise). We have been making use of the oom killer to save a system from irreconcilable overcommit situations.

---

---

Subject: Re: [PATCH v2 02/13] memcg: Kernel memory accounting infrastructure.  
Posted by [Glauber Costa](#) on Tue, 13 Mar 2012 17:31:40 GMT  
[View Forum Message](#) <> [Reply to Message](#)

---

On 03/13/2012 09:00 PM, Greg Thelen wrote:

> Glauber Costa <glommer@parallels.com> writes:

>> 2) For the kernel itself, we are mostly concerned that a malicious container may  
>> pin into memory big amounts of kernel memory which is, ultimately,  
>> unreclaimable. In particular, with overcommit allowed scenarios, you can fill  
>> the whole physical memory (or at least a significant part) with those objects,  
>> well beyond your softlimit allowance, making the creation of further containers  
>> impossible.

>> With user memory, you can reclaim the cgroup back to its place. With kernel  
>> memory, you can't.

>

> In overcommit situations the page allocator starts failing even though  
> memcg page can charge pages.

If you overcommit mem+swap, yes. If you overcommit mem, no: reclaim happens first. And we don't have that option with pinned kernel memory.

Of course you \*can\* run your system without swap, but the whole thing exists exactly because there is a large enough # of ppl who wants to be able to overcommit their physical memory, without failing allocations.

---

---

Subject: Re: [PATCH v2 02/13] memcg: Kernel memory accounting infrastructure.  
Posted by [KAMEZAWA Hiroyuki](#) on Wed, 14 Mar 2012 00:15:26 GMT  
[View Forum Message](#) <> [Reply to Message](#)

---

On Tue, 13 Mar 2012 14:37:30 +0400

Glauber Costa <glommer@parallels.com> wrote:

> > After looking codes, I think we need to think  
> > whether independent\_kmem\_limit is good or not....

> >

> > How about adding MEMCG\_KMEM\_ACCOUNT flag instead of this and use only  
> > memcg->res/memcg->memsw rather than adding a new counter, memcg->kmem ?

> >

> > if MEMCG\_KMEM\_ACCOUNT is set -> slab is accounted to mem->res/memsw.

> > if MEMCG\_KMEM\_ACCOUNT is not set -> slab is never accounted.

> >

> > (I think On/Off switch is required..)  
> >  
> > Thanks,  
> > -Kame  
> >  
>  
> This has been discussed before, I can probably find it in the archives  
> if you want to go back and see it.  
>

Yes. IIUC, we agreed to have independent kmem limit. I just want to think it again because there are too many proposals and it seems I'm in confusion.

As far as I see, there are ongoing works as

- kmem limit by 2 guys.
- hugetlb limit
- per lru locking (by 2 guys)
- page cgroup diet (by me, but stops now.)
- dirty-ratio and writeback
- Tejun's proposal to remove pre\_destroy()
- moving shared resource

I'm thinking what is a simple plan and implementation.  
Most of series consists of 10+ patches...

Thank you for your help of clarification.

> But in a nutshell:

>  
> 1) Supposing independent knob disappear (I will explain in item 2 why I  
> don't want it to), I don't think a flag makes sense either. \*If\* we are  
> planning to enable/disable this, it might make more sense to put some  
> work on it, and allow particular slabs to be enabled/disabled by writing  
> to memory.kmem.slabinfo (-\* would disable all, +\* enable all, +kmallocc\*  
> enable all kmallocc, etc).

>  
seems interesting.

> Alternatively, what we could do instead, is something similar to what  
> ended up being done for tcp, by request of the network people: if you  
> never touch the limit file, don't bother with it at all, and simply does  
> not account. With Suleiman's lazy allocation infrastructure, that should  
> actually be trivial. And then again, a flag is not necessary, because  
> writing to the limit file does the job, and also convey the meaning well  
> enough.

>

Hm.

- > 2) For the kernel itself, we are mostly concerned that a malicious
- > container may pin into memory big amounts of kernel memory which is,
- > ultimately, unreclaimable.

Yes. This is a big problem both to memcg and the whole system.

In my experience, 2000 process shares a 10GB shared memory and eats up big memory ;(

- > In particular, with overcommit allowed
- > scenarios, you can fill the whole physical memory (or at least a
- > significant part) with those objects, well beyond your softlimit
- > allowance, making the creation of further containers impossible.
- > With user memory, you can reclaim the cgroup back to its place. With
- > kernel memory, you can't.

>  
Agreed.

- > In the particular example of 32-bit boxes, you can easily fill up a
- > large part of the available 1gb kernel memory with pinned memory and
- > render the whole system unresponsive.
- >
- > Never allowing the kernel memory to go beyond the soft limit was one of
- > the proposed alternatives. However, it may force you to establish a soft
- > limit where one was not previously needed. Or, establish a low soft
- > limit when you really need a bigger one.
- >
- > All that said, while reading your message, thinking a bit, the following
- > crossed my mind:
- >
- > - We can account the slabs to memcg->res normally, and just store the
- > information that this is kernel memory into a percpu counter, as
- > I proposed recently.

Ok, then user can see the amount of kernel memory.

- > - The knob goes away, and becomes implicit: if you ever write anything
- > to memory.kmem.limit\_in\_bytes, we transfer that memory to a separate
- > kmem res\_counter, and proceed from there. We can keep accounting to
- > memcg->res anyway, just that kernel memory will now have a separate
- > limit.

Okay, then,

`kmem_limit < memory.limit < memsw.limit`

...seems reasonable to me.

This means, user can specify 'ratio' of kmem in `memory.limit`.

More consideration will be interesting.

- We can show the amount of reclaimable kmem by some means ?
- What happens when a new cgroup created ?
- Should we have 'ratio' interface in kernel level ?
- What happens at task moving ?
- Should we allow per-slab accounting knob in `/sys/kernel/slab/xxx` ? or somewhere ?
- Should we show per-memcg usage in `/sys/kernel/slab/xxx` ?
- Should we have `force_empty` for kmem (as last resort) ?

With any implementation, my concern is

- overhead/performance.
- unreclaimable kmem
- shared kmem between cgroups.

> - With this scheme, it may not be necessary to ever have a file  
> `memory.kmem.soft_limit_in_bytes`. Reclaim is always part of the normal  
> memcg reclaim.

>  
Good.

> The outlined above would work for us, and make the whole scheme simpler,  
> I believe.

>  
> What do you think ?

It sounds interesting to me.

Thanks,  
-Kame

---

Subject: Re: [PATCH v2 02/13] memcg: Kernel memory accounting infrastructure.  
Posted by [Glauber Costa](#) on Wed, 14 Mar 2012 12:29:25 GMT  
[View Forum Message](#) <> [Reply to Message](#)

>> This has been discussed before, I can probably find it in the archives  
>> if you want to go back and see it.  
>>

>  
> Yes. IIUC, we agreed to have independent kmem limit. I just want to think it  
> again because there are too many proposals and it seems I'm in confusion.  
>

Sure thing. The discussion turned out good, so I'm glad you asked =)

>  
>> But in a nutshell:  
>>  
>> 1) Supposing independent knob disappear (I will explain in item 2 why I  
>> don't want it to), I don't think a flag makes sense either. \*If\* we are  
>> planning to enable/disable this, it might make more sense to put some  
>> work on it, and allow particular slabs to be enabled/disabled by writing  
>> to memory.kmem.slabinfo (-\* would disable all, +\* enable all, +kmallocc\*  
>> enable all kmalloc, etc).  
>>  
> seems interesting.  
> I'll try to cook a PoC.

>> All that said, while reading your message, thinking a bit, the following  
>> crossed my mind:

>>  
>> - We can account the slabs to memcg->res normally, and just store the  
>> information that this is kernel memory into a percpu counter, as  
>> I proposed recently.

>  
> Ok, then user can see the amount of kernel memory.

>  
>  
>> - The knob goes away, and becomes implicit: if you ever write anything  
>> to memory.kmem.limit\_in\_bytes, we transfer that memory to a separate  
>> kmem res\_counter, and proceed from there. We can keep accounting to  
>> memcg->res anyway, just that kernel memory will now have a separate  
>> limit.

>  
> Okay, then,  
>  
> kmem\_limit< memory.limit< memsw.limit

>  
> ...seems reasonable to me.  
> This means, user can specify 'ratio' of kmem in memory.limit.  
> Yes, I believe so. It is a big improvement over the current interface  
> we have today, IMHO.

>  
> More consideration will be interesting.  
>

> - We can show the amount of reclaimable kmem by some means ?  
That's hard to do. The users of the cache have this information, the underlying slab/slub/slut code do not. We need to rely on the cache owner to provide this, and provide correctly. So the chances we'll have incorrect information here grows by quite a bit.

> - What happens when a new cgroup created ?

mem\_cgroup\_create() is called =)

Heh, jokes apart, I don't really follow here. What exactly do you mean? There shouldn't be anything extremely out of the ordinary.

> - Should we have 'ratio' interface in kernel level ?

I personally don't like a ratio interface. I believe specifying "kmem should never be allowed to go over X bytes" is more than enough.

> - What happens at task moving ?

From kmem PoV, nothing. It is ultimately impossible to track a slab page to a task. The page contains objects that were allocated from multiple tasks. Only when the whole cgroup is destroyed, is that we take any action.

> - Should we allow per-slab accounting knob in /sys/kernel/slab/xxx ?  
> or somewhere ?

Not really follow.

> - Should we show per-memcg usage in /sys/kernel/slab/xxx ?  
I guess so.

> - Should we have force\_empty for kmem (as last resort) ?

We do that when the cgroup is going away. From user action, I suspect the best we can do is call the shrinkers, and see if they get freed.

>

> With any implementation, my concern is

> - overhead/performance.

Yes. For the next round, we need to add some more detailed benchmarks.

> - unreclaimable kmem

That's actually the reason behind all that!

So if you have a 1 Gb mem allowance, and you fill it with unreclaimable kmem, you are in trouble, yes.

Point is, at least all your allocations will stop working, and something will be done soon. Without kmem tracking, this can grow and grow, outside the cgroups border.





> More majordomo info at <http://vger.kernel.org/majordomo-info.html>

---

---

Subject: Re: [PATCH v2 02/13] memcg: Kernel memory accounting infrastructure.  
Posted by [KAMEZAWA Hiroyuki](#) on Thu, 15 Mar 2012 00:48:27 GMT  
[View Forum Message](#) <> [Reply to Message](#)

---

(2012/03/14 21:29), Glauber Costa wrote:

>> - What happens when a new cgroup created ?  
>  
> mem\_cgroup\_create() is called =)  
> Heh, jokes apart, I don't really follow here. What exactly do you mean?  
> There shouldn't be anything extremely out of the ordinary.  
>

Sorry, too short words.

Assume a cgroup with  
cgroup.memory.limit\_in\_bytes=1G  
cgroup.memory.kmem.limit\_in\_bytes=400M

When a child cgroup is created, what should be the default values.  
'unlimited' as current implementation ?  
Hmm..maybe yes.

Thanks,  
-Kame

---

---

Subject: Re: [PATCH v2 02/13] memcg: Kernel memory accounting infrastructure.  
Posted by [Glauber Costa](#) on Thu, 15 Mar 2012 11:07:15 GMT  
[View Forum Message](#) <> [Reply to Message](#)

---

On 03/15/2012 04:48 AM, KAMEZAWA Hiroyuki wrote:

>>> - What happens when a new cgroup created ?  
>> >  
>> > mem\_cgroup\_create() is called =)  
>> > Heh, jokes apart, I don't really follow here. What exactly do you mean?  
>> > There shouldn't be anything extremely out of the ordinary.  
>> >  
>  
> Sorry, too short words.  
>  
> Assume a cgroup with

> cgroup.memory.limit\_in\_bytes=1G  
> cgroup.memory.kmem.limit\_in\_bytes=400M  
>  
> When a child cgroup is created, what should be the default values.  
> 'unlimited' as current implementation ?  
> Hmm..maybe yes.

I think so, yes. I see no reason to come up with any default values in memcg. Yes, your allocations can fail due to your parent limits. But since I never heard of any machine with 9223372036854775807 bytes of memory, that is true even for the root memcg =)

---

---

Subject: Re: [PATCH v2 02/13] memcg: Kernel memory accounting infrastructure.  
Posted by [Peter Zijlstra](#) on Thu, 15 Mar 2012 11:13:34 GMT  
[View Forum Message](#) <> [Reply to Message](#)

---

On Thu, 2012-03-15 at 15:07 +0400, Glauber Costa wrote:  
> But since I never heard of any machine with  
> 9223372036854775807 bytes of memory, that is true even for the root memcg

What, you don't have more than 8 exabyte of memory in your laptop !?  
Surely you're due for an upgrade then.

---

---

Subject: Re: [PATCH v2 02/13] memcg: Kernel memory accounting infrastructure.  
Posted by [Glauber Costa](#) on Thu, 15 Mar 2012 11:21:23 GMT  
[View Forum Message](#) <> [Reply to Message](#)

---

On 03/15/2012 03:13 PM, Peter Zijlstra wrote:  
> On Thu, 2012-03-15 at 15:07 +0400, Glauber Costa wrote:  
>> But since I never heard of any machine with  
>> 9223372036854775807 bytes of memory, that is true even for the root memcg  
>  
> What, you don't have more than 8 exabyte of memory in your laptop !?  
> Surely you're due for an upgrade then.

Yeah, I requested it already, but I was told it could take a while

---