Subject: Re: [ckrm-tech] [RFC][PATCH][2/4] Add RSS accounting and control
Posted by Balbir Singh on Mon, 19 Feb 2007 11:09:33 GMT
View Forum Message <> Reply to Message

Andrew Morton wrote:
> On Mon, 19 Feb 2007 16:07:44 +0530 Balbir Singh <balbir@in.ibm.com> wrote:
>
>>>> +void memctlr_mm_free(struct mm_struct *mm)
>>>> +{
>>>> + kfree(mm->counter);
>>>> +}
>>>> +
>>>> +static inline void memctlr_mm_assign_container_direct(struct mm_struct *mm,
>>>> + struct container *cont)
>>>> +{
>>>> + write_lock(&mm->container_lock);
>>>> + mm->container = cont;
>>>> + write_unlock(&mm->container_lock);
>>>> +}
>>> More weird locking here.
>>>
>> The container field of the mm_struct is protected by a read write spin lock.
>
> That doesn't mean anything to me.
>
> What would go wrong if the above locking was simply removed? And how does
> the locking prevent that fault?
>

Some pages could charged to the wrong container. Apart from that I do not
see anything going bad (I'll double check that).

>
>>>> +void memctlr_mm_assign_container(struct mm_struct *mm, struct task_struct *p)
>>>> +{
>>>> + struct container *cont = task_container(p, &memctlr_subsys);
>>>> + struct memctlr *mem = memctlr_from_cont(cont);
>>>> +
>>>> + BUG_ON(!mem);
>>>> + write_lock(&mm->container_lock);
>>>> + mm->container = cont;
>>>> + write_unlock(&mm->container_lock);
>>>> +}
>>> And here.
>> Ditto.
>
> ditto ;)
>

:-)

>>>> +/*
>>>> + * Update the rss usage counters for the mm_struct and the container it belongs
>>>> + * to. We do not fail rss for pages shared during fork (see copy_one_pte()).
>>>> + */
>>>> +int memctlr_update_rss(struct mm_struct *mm, int count, bool check)
>>>> +{
>>>> + int ret = 1;
>>>> + struct container *cont;
>>>> + long usage, limit;
>>>> + struct memctlr *mem;
>>>> +
>>>> + read_lock(&mm->container_lock);
>>>> + cont = mm->container;
>>>> + read_unlock(&mm->container_lock);
>>>> +
>>>> + if (!cont)
>>>> + goto done;
>>> And here. I mean, if there was a reason for taking the lock around that
>>> read, then testing `cont' outside the lock just invalidated that reason.
>>>
>> We took a consistent snapshot of cont. It cannot change outside the lock,
>> we check the value outside. I am sure I missed something.
>
> If it cannot change outside the lock then we don't need to take the lock!
>

We took a snapshot that we thought was consistent. We check for the value
outside. I guess there is no harm, the worst thing that could happen
is wrong accounting during mm->container changes (when a task changes
container).

>> MEMCTLR_DONT_CHECK_LIMIT exists for the following reasons
>>
>> 1. Pages are shared during fork, fork() is not failed at that point
>>   since the pages are shared anyway, we allow the RSS limit to be
>>   exceeded.
>> 2. When ZERO_PAGE is added, we don't check for limits (zeromap_pte_range).
>> 3. On reducing RSS (passing -1 as the value)
>
> OK, that might make a nice comment somewhere (if it's not already there).

Yes, thanks for keeping us humble and honest, I'll add it.

--
 Warm Regards,

Subject: Re: [ckrm-tech] [RFC][PATCH][2/4] Add RSS accounting and control
Posted by Andrew Morton on Mon, 19 Feb 2007 11:23:52 GMT
View Forum Message <> Reply to Message

On Mon, 19 Feb 2007 16:39:33 +0530 Balbir Singh <balbir@in.ibm.com> wrote:

> Andrew Morton wrote:
> > On Mon, 19 Feb 2007 16:07:44 +0530 Balbir Singh <balbir@in.ibm.com> wrote:
> >
> >>>> +void memctlr_mm_free(struct mm_struct *mm)
> >>>> +{
> >>>> + kfree(mm->counter);
> >>>> +}
> >>>> +
> >>>> +static inline void memctlr_mm_assign_container_direct(struct mm_struct *mm,
> >>>> + struct container *cont)
> >>>> +{
> >>>> + write_lock(&mm->container_lock);
> >>>> + mm->container = cont;
> >>>> + write_unlock(&mm->container_lock);
> >>>> +}
> >>> More weird locking here.
> >>>
> >> The container field of the mm_struct is protected by a read write spin lock.
> >
> > That doesn't mean anything to me.
> >
> > What would go wrong if the above locking was simply removed? And how does
> > the locking prevent that fault?
> >
>
> Some pages could charged to the wrong container. Apart from that I do not
> see anything going bad (I'll double check that).

Argh. Please, think about this.

That locking *doesn't do anything*. Except for that one situation I
described: some other holder of the lock reads mm->container twice inside
the lock and requires that the value be the same both times (and that sort
of code should be converted to take a local copy, so this locking here can
be removed).

>>>> +
>>>> + read_lock(&mm->container_lock);
>>>> + cont = mm->container;

> >>>> + read_unlock(&mm->container_lock);
> >>>> +
> >>>> + if (!cont)
> >>>> + goto done;
> >>> And here. I mean, if there was a reason for taking the lock around that
> >>> read, then testing `cont' outside the lock just invalidated that reason.
> >>>
> >> We took a consistent snapshot of cont. It cannot change outside the lock,
> >> we check the value outside. I am sure I missed something.
> >
> > If it cannot change outside the lock then we don't need to take the lock!
> >
>
> We took a snapshot that we thought was consistent.

Consistent with what? That's a single-word read inside that lock.

> We check for the value
> outside. I guess there is no harm, the worst thing that could happen
> is wrong accounting during mm->container changes (when a task changes
> container).

If container->lock is held when a task is removed from the
container then yes, `cont' here can refer to a container to which the task
no longer belongs.

More worrisome is the potential for use-after-free. What prevents the
pointer at mm->container from referring to freed memory after we're dropped
the lock?

Subject: Re: [ckrm-tech] [RFC][PATCH][2/4] Add RSS accounting and control
Posted by Balbir Singh on Mon, 19 Feb 2007 11:56:31 GMT
View Forum Message <> Reply to Message

Andrew Morton wrote:
> On Mon, 19 Feb 2007 16:39:33 +0530 Balbir Singh <balbir@in.ibm.com> wrote:
>
>> Andrew Morton wrote:
>>> On Mon, 19 Feb 2007 16:07:44 +0530 Balbir Singh <balbir@in.ibm.com> wrote:
>>>
>>>>> +void memctlr_mm_free(struct mm_struct *mm)
>>>>> +{
>>>>> + kfree(mm->counter);
>>>>> +}
>>>>> +
>>>>> +static inline void memctlr_mm_assign_container_direct(struct mm_struct *mm,
>>>>> +   struct container *cont)

>>>>>> +{
>>>>>> + write_lock(&mm->container_lock);
>>>>>> + mm->container = cont;
>>>>>> + write_unlock(&mm->container_lock);
>>>>>> +}
>>>>> More weird locking here.
>>>>>
>>>> The container field of the mm_struct is protected by a read write spin lock.
>>> That doesn't mean anything to me.
>>>
>>> What would go wrong if the above locking was simply removed? And how does
>>> the locking prevent that fault?
>>>
>> Some pages could charged to the wrong container. Apart from that I do not
>> see anything going bad (I'll double check that).
>
> Argh. Please, think about this.
>

Sure, I will. I guess I am short circuiting my thinking process :-)

> That locking *doesn't do anything*. Except for that one situation I
> described: some other holder of the lock reads mm->container twice inside
> the lock and requires that the value be the same both times (and that sort
> of code should be converted to take a local copy, so this locking here can
> be removed).
>

Yes, that makes sense.

>>>>>> +
>>>>>> + read_lock(&mm->container_lock);
>>>>>> + cont = mm->container;
>>>>>> + read_unlock(&mm->container_lock);
>>>>>> +
>>>>>> + if (!cont)
>>>>>> + goto done;
>>>>> And here. I mean, if there was a reason for taking the lock around that
>>>>> read, then testing `cont' outside the lock just invalidated that reason.
>>>>>
>>>> We took a consistent snapshot of cont. It cannot change outside the lock,
>>>> we check the value outside. I am sure I missed something.
>>> If it cannot change outside the lock then we don't need to take the lock!
>>>
>> We took a snapshot that we thought was consistent.
>
> Consistent with what? That's a single-word read inside that lock.

>

Yes, that makes sense.

>> We check for the value
>> outside. I guess there is no harm, the worst thing that could happen
>> is wrong accounting during mm->container changes (when a task changes
>> container).
>
> If container->lock is held when a task is removed from the
> container then yes, `cont' here can refer to a container to which the task
> no longer belongs.
>
> More worrisome is the potential for use-after-free. What prevents the
> pointer at mm->container from referring to freed memory after we're dropped
> the lock?
>

The container cannot be freed unless all tasks holding references to it are
gone, that would ensure that all mm->containers are pointing elsewhere and
never to a stale value.

I hope my short-circuited brain got this right :-)

--
Warm Regards,
Balbir Singh

Subject: Re: [ckrm-tech] [RFC][PATCH][2/4] Add RSS accounting and control
Posted by Paul Menage on Mon, 19 Feb 2007 12:09:38 GMT
View Forum Message <> Reply to Message

On 2/19/07, Balbir Singh <balbir@in.ibm.com> wrote:
> >
> > More worrisome is the potential for use-after-free. What prevents the
> > pointer at mm->container from referring to freed memory after we're dropped
> > the lock?
> >
>
> The container cannot be freed unless all tasks holding references to it are
> gone,

... or have been moved to other containers. If you're not holding
task->alloc_lock or one of the container mutexes, there's nothing to
stop the task being moved to another container, and the container

being deleted.

If you're in an RCU section then you can guarantee that the container
(that you originally read from the task) and its subsystems at least
won't be deleted while you're accessing them, but for accounting like
this I suspect that's not enough, since you need to be adding to the
accounting stats on the correct container. I think you'll need to hold
mm->container_lock for the duration of memctl_update_rss()

Paul

Subject: Re: [ckrm-tech] [RFC][PATCH][2/4] Add RSS accounting and control
Posted by Balbir Singh on Mon, 19 Feb 2007 14:10:06 GMT
View Forum Message <> Reply to Message

Paul Menage wrote:
> On 2/19/07, Balbir Singh <balbir@in.ibm.com> wrote:
>>> More worrisome is the potential for use-after-free. What prevents the
>>> pointer at mm->container from referring to freed memory after we're dropped
>>> the lock?
>>>
>> The container cannot be freed unless all tasks holding references to it are
>> gone,
>
> ... or have been moved to other containers. If you're not holding
> task->alloc_lock or one of the container mutexes, there's nothing to
> stop the task being moved to another container, and the container
> being deleted.
>
> If you're in an RCU section then you can guarantee that the container
> (that you originally read from the task) and its subsystems at least
> won't be deleted while you're accessing them, but for accounting like
> this I suspect that's not enough, since you need to be adding to the
> accounting stats on the correct container. I think you'll need to hold
> mm->container_lock for the duration of memctl_update_rss()
>
> Paul
>

Yes, that sounds like the correct thing to do.

--
Warm Regards,
Balbir Singh

Subject: Re: [ckrm-tech] [RFC][PATCH][2/4] Add RSS accounting and control
Posted by Vaidyanathan Srinivas on Mon, 19 Feb 2007 16:07:31 GMT
View Forum Message <> Reply to Message

Balbir Singh wrote:
> Paul Menage wrote:
>> On 2/19/07, Balbir Singh <balbir@in.ibm.com> wrote:
>>>> More worrisome is the potential for use-after-free. What prevents the
>>>> pointer at mm->container from referring to freed memory after we're dropped
>>>> the lock?
>>>>
>>> The container cannot be freed unless all tasks holding references to it are
>>> gone,
>> ... or have been moved to other containers. If you're not holding
>> task->alloc_lock or one of the container mutexes, there's nothing to
>> stop the task being moved to another container, and the container
>> being deleted.
>>
>> If you're in an RCU section then you can guarantee that the container
>> (that you originally read from the task) and its subsystems at least
>> won't be deleted while you're accessing them, but for accounting like
>> this I suspect that's not enough, since you need to be adding to the
>> accounting stats on the correct container. I think you'll need to hold
>> mm->container_lock for the duration of memctl_update_rss()
>>
>> Paul
>>
>
> Yes, that sounds like the correct thing to do.
>

Accounting accuracy will anyway be affected when a process is migrated
while it is still allocating pages. Having a lock here does not
necessarily improve the accounting accuracy. Charges from the old
container would have to be moved to the new container before deletion
which implies all tasks have already left the container and no
mm_struct is holding a pointer to it.

The only condition that will break our code will be if the container
pointer becomes invalid while we are updating stats. This can be
prevented by RCU section as mentioned by Paul. I believe explicit
lock and unlock may not provide additional benefit here.

--Vaidy