Subject: Re: [BUGFIX][RFC][PATCH][only -mm] FIX memory leak in memory cgroup vs. page migration [1/1] fix pag
Posted by Balbir Singh on Tue, 02 Oct 2007 15:34:40 GMT
View Forum Message <> Reply to Message

KAMEZAWA Hiroyuki wrote:
> While using memory control cgroup, page-migration under it works as following.
> ==
> 1. uncharge all refs at try to unmap.
> 2. charge regs again remove_migration_ptes()
> ==
> This is simple but has following problems.
> ==
> The page is uncharged and chaged back again if *mapped*.
> - This means that cgroup before migraion can be different from one after
> migraion

>From the test case mentioned earlier, this happens because the task has moved from one cgroup to another, right?

> - If page is not mapped but charged as page cache, charge is just ignored
> (because not mapped, it will not be uncharged before migration)

OK. This is an interesting situation, we uncharge page cache only in __remove_from_page_cache(). This is a combination of task migration followed by page migration, which the memory controller does not handle very well at the moment.

> This is memory leak.

Yes, it is.

> ==
> This is bad.

Absolutely!

> And migration can migrate *not mapped* pages in future by migration-by-kernel
> driven by memory-unplug and defragment-by-migration at el.
>
> This patch tries to keep memory cgroup at page migration by increasing
> one refcnt during it. 3 functions are added.
> mem_cgroup_prepare_migration() --- increase refcnt of page->page_cgroup
> mem_cgroup_end_migration() --- decrease refcnt of page->page_cgroup
> mem_cgroup_page_migration() --- copy page->page_cgroup from old page to
> new page.
>
> Obviously, mem_cgroup_isolate_pages() and this page migration, which

> copies page_cgroup from old page to new page, has race.
>
> There seem to be 3 ways for avoiding this race.
> A. take mem_group->lock while mem_cgroup_page_migration().
> B. isolate pc from mem_cgroup's LRU when we isolate page from zone's LRU.
> C. ignore non-LRU page at mem_cgroup_isolate_pages().
>
> This patch uses method (C.) and modifes mem_cgroup_isolate_pages() igonres
> !PageLRU pages.
>

The page(s) is(are) !PageLRU only during page migration right?

> Tested and worked well in ia64/NUMA box.
>
> Signed-off-by: KAMEZAWA Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
>
> ---
> include/linux/memcontrol.h | 22 ++++++++++++++++
> mm/memcontrol.c | 62 ++---
> mm/migrate.c | 13 +++++++--
> 3 files changed, 90 insertions(+), 7 deletions(-)
>
> Index: linux-2.6.23-rc8-mm2/include/linux/memcontrol.h
> ===
> --- linux-2.6.23-rc8-mm2.orig/include/linux/memcontrol.h
> +++ linux-2.6.23-rc8-mm2/include/linux/memcontrol.h
> @@ -43,8 +43,14 @@ extern unsigned long mem_cgroup_isolate_
>   struct mem_cgroup *mem_cont,
>   int active);
> extern void mem_cgroup_out_of_memory(struct mem_cgroup *mem, gfp_t gfp_mask);
> +
> extern int mem_cgroup_cache_charge(struct page *page, struct mm_struct *mm,
>   gfp_t gfp_mask);
> +/* For handling page migration in proper way */
> +extern void mem_cgroup_prepare_migration(struct page *page);
> +extern void mem_cgroup_end_migration(struct page *page);
> +extern void mem_cgroup_page_migration(struct page *newpage, struct page *page);
> +
>
> static inline struct mem_cgroup *mm_cgroup(const struct mm_struct *mm)
> {
> @@ -107,6 +113,22 @@ static inline struct mem_cgroup *mm_cgro
>   return NULL;
> }
>
> +/* For page migration */
> +static inline void mem_cgroup_prepare_migration(struct page *page)

```
> +{
> + return;
> +}
> +
> +static inline void mem_cgroup_end_migration(struct mem_cgroup *cgroup)
> +{
> + return;
> +}
> +
> +static inline void
> +mem_cgroup_page_migration(struct page *newpage, struct page *page)
> +{
> + return;
> +}
>  #endif /* CONFIG_CGROUP_MEM_CONT */
>
>  #endif /* _LINUX_MEMCONTROL_H */
> Index: linux-2.6.23-rc8-mm2/mm/memcontrol.c
> ===================================================================
> --- linux-2.6.23-rc8-mm2.orig/mm/memcontrol.c
> +++ linux-2.6.23-rc8-mm2/mm/memcontrol.c
> @@ -198,7 +198,7 @@ unsigned long mem_cgroup_isolate_pages(u
>   unsigned long scan;
>   LIST_HEAD(pc_list);
>   struct list_head *src;
> - struct page_cgroup *pc;
> + struct page_cgroup *pc, *tmp;
>
>   if (active)
>    src = &mem_cont->active_list;
> @@ -206,8 +206,10 @@ unsigned long mem_cgroup_isolate_pages(u
>    src = &mem_cont->inactive_list;
>
>   spin_lock(&mem_cont->lru_lock);
> - for (scan = 0; scan < nr_to_scan && !list_empty(src); scan++) {
> -  pc = list_entry(src->prev, struct page_cgroup, lru);
> + scan = 0;
> + list_for_each_entry_safe_reverse(pc, tmp, src, lru) {
> +  if (scan++ >= nr_to_scan)
> +   break;
>    page = pc->page;
>    VM_BUG_ON(!pc);
>
> @@ -225,9 +227,14 @@ unsigned long mem_cgroup_isolate_pages(u
>    /*
>     * Reclaim, per zone
>     * TODO: make the active/inactive lists per zone
> +   * !PageLRU page can be found under us while migration.
```

> +   * just ignore it.
>   */
> - if (page_zone(page) != z)
> + if (page_zone(page) != z || !PageLRU(page)) {

I would prefer to do unlikely(!PageLRU(page)), since most of the
times the page is not under migration

> +   /* Skip this */
> +   /* Don't decrease scan here for avoiding dead lock */

Could we merge the two comments to one block comment?

>   continue;
> + }
>
>   /*
>   * Check if the meta page went away from under us
> @@ -417,8 +424,14 @@ void mem_cgroup_uncharge(struct page_cgr
>   return;
>
>   if (atomic_dec_and_test(&pc->ref_cnt)) {
> +retry:
>   page = pc->page;
>   lock_page_cgroup(page);
> + /* migration occur ? */
> + if (page_get_page_cgroup(page) != pc) {
> +   unlock_page_cgroup(page);
> +   goto retry;

Shouldn't we check if page_get_page_cgroup(page) returns
NULL, if so, unlock and return?

> + }
>   mem = pc->mem_cgroup;
>   css_put(&mem->css);
>   page_assign_page_cgroup(page, NULL);
> @@ -432,6 +445,47 @@ void mem_cgroup_uncharge(struct page_cgr
>   }
> }
>
> +void mem_cgroup_prepare_migration(struct page *page)
> +{
> + struct page_cgroup *pc;
> + lock_page_cgroup(page);
> + pc = page_get_page_cgroup(page);
> + if (pc)
> + atomic_inc(&pc->ref_cnt);

> + unlock_page_cgroup(page);
> + return;
> +}
> +
> +void mem_cgroup_end_migration(struct page *page)
> +{
> + struct page_cgroup *pc = page_get_page_cgroup(page);
> + mem_cgroup_uncharge(pc);
> +}
> +
> +void mem_cgroup_page_migration(struct page *newpage, struct page *page)
> +{
> + struct page_cgroup *pc;
> + /*
> + * We already keep one reference to paage->cgroup.
> + * and both pages are guaranteed to be locked under page migration.
> + */
> + lock_page_cgroup(page);
> + lock_page_cgroup(newpage);
> + pc = page_get_page_cgroup(page);
> + if (!pc)
> + goto unlock_out;
> + page_assign_page_cgroup(page, NULL);
> + pc->page = newpage;
> + page_assign_page_cgroup(newpage, pc);
> +
> +unlock_out:
> + unlock_page_cgroup(newpage);
> + unlock_page_cgroup(page);
> + return;
> +}
> +
> +
> +
> int mem_cgroup_write_strategy(char *buf, unsigned long long *tmp)
> {
> *tmp = memparse(buf, &buf);
> Index: linux-2.6.23-rc8-mm2/mm/migrate.c
> ===
> --- linux-2.6.23-rc8-mm2.orig/mm/migrate.c
> +++ linux-2.6.23-rc8-mm2/mm/migrate.c
> @@ -598,9 +598,10 @@ static int move_to_new_page(struct page
> else
> rc = fallback_migrate_page(mapping, newpage, page);
>
> - if (!rc)
> + if (!rc) {
> + mem_cgroup_page_migration(newpage, page);

>   remove_migration_ptes(page, newpage);
> - else
> + } else
>   newpage->mapping = NULL;
>
>   unlock_page(newpage);
> @@ -651,6 +652,8 @@ static int unmap_and_move(new_page_t get
>   rcu_read_lock();
>   rcu_locked = 1;
>   }
> + mem_cgroup_prepare_migration(page);
> +
>   /*
>   * This is a corner case handling.
>   * When a new swap-cache is read into, it is linked to LRU
> @@ -666,8 +669,12 @@ static int unmap_and_move(new_page_t get
>   if (!page_mapped(page))
>   rc = move_to_new_page(newpage, page);
>
> - if (rc)
> + if (rc) {
>   remove_migration_ptes(page, page);
> + mem_cgroup_end_migration(page);
> + } else
> + mem_cgroup_end_migration(newpage);
> +
> rcu_unlock:
>   if (rcu_locked)
>   rcu_read_unlock();
>

Looks good so far, even though I am yet to test. It looks like a tough
problem to catch and debug. Thanks!

--
Warm Regards,
Balbir Singh
Linux Technology Center
IBM, ISTL