
Subject: Re: [RFC 3/4] memcg: background reclaim
Posted by [Balbir Singh](#) on Tue, 27 May 2008 17:08:09 GMT
[View Forum Message](#) <> [Reply to Message](#)

KAMEZAWA Hiroyuki wrote:

> Do background reclaim based on high-low watermarks.
>
> This feature helps smooth work of processes under memcg by reclaiming memory
> in the kernel thread. # of limitation failure at mem_cgroup_charge() will
> dramatically reduced. But this also means a CPU is continuously used for
> reclaiming memory.
>
> This one is very simple. Anyway, we need to update this when we add new
> complexity to memcg.
>
> Major logic:
> - add high-low watermark support to memory resource controller.
> - create a kernel thread for cgroup when hwmark is changed. (once)
> - stop a kernel thread at rmdir().
> - start background reclaim if res_counter is over high-watermark.
> - stop background reclaim if res_coutner is below low-watermark.
> - for reclaiming, just calls try_to_free_mem_cgroup_pages().
> - kthread for reclaim 's priority is nice(0). default is (-5).
> (weaker is better ?)
> - kthread for reclaim calls yield() on each loop.
>
> TODO:
> - add an interface to start/stop daemon ?
> - wise numa support
> - too small "low watermark" targe just consumes CPU. Should we warn ?
> and what is the best value for hwmark/lwmark in general....?
>
> Changelog: old one -> this (v1)
> - start a thread at write of hwmark.
>
>
> Signed-off-by: KAMEZAWA Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
>
> ---
> mm/memcontrol.c | 147 +++-----
> 1 file changed, 130 insertions(+), 17 deletions(-)
>
> Index: mm-2.6.26-rc2-mm1/mm/memcontrol.c
> =====
> --- mm-2.6.26-rc2-mm1.orig/mm/memcontrol.c
> +++ mm-2.6.26-rc2-mm1/mm/memcontrol.c
> @@ -32,7 +32,8 @@
> #include <linux/fs.h>

```

> #include <linux/seq_file.h>
> #include <linux/vmalloc.h>
> -
> +#include <linux/freezer.h>
> +#include <linux/kthread.h>
> #include <asm/uaccess.h>
>
> struct cgroup_subsys mem_cgroup_subsys __read_mostly;
> @@ -119,10 +120,6 @@ struct mem_cgroup_lru_info {
> * statistics based on the statistics developed by Rik Van Riel for clock-pro,
> * to help the administrator determine what knobs to tune.
> *
> - * TODO: Add a water mark for the memory controller. Reclaim will begin when
> - * we hit the water mark. May be even add a low water mark, such that
> - * no reclaim occurs from a cgroup at it's low water mark, this is
> - * a feature that will be implemented much later in the future.
> */
> struct mem_cgroup {
> struct cgroup_subsys_state css;
> @@ -131,6 +128,13 @@ struct mem_cgroup {
> */
> struct res_counter res;
> /*
> + * background reclaim.
> + */
> + struct {
> + wait_queue_head_t waitq;
> + struct task_struct *thread;
> + } daemon;

```

Comments on each of the members would be nice.

```

> + /*
> * Per cgroup active and inactive list, similar to the
> * per zone LRU lists.
> */
> @@ -143,6 +147,7 @@ struct mem_cgroup {
> struct mem_cgroup_stat stat;
> };
> static struct mem_cgroup init_mem_cgroup;
> +static DEFINE_MUTEX(memcont_daemon_lock);
>
> /*
> * We use the lower bit of the page->page_cgroup pointer as a bit spin
> @@ -374,6 +379,15 @@ void mem_cgroup_move_lists(struct page *
> unlock_page_cgroup(page);
> }
>

```

```

> +static void mem_cgroup_schedule_reclaim(struct mem_cgroup *mem)
> +{
> + if (!mem->daemon.thread)
> + return;

```

I suspect we are using threads. Aren't workqueues better than threads?

```

> + if (!waitqueue_active(&mem->daemon.waitq))
> + return;
> + wake_up_interruptible(&mem->daemon.waitq);
> +}
> +
> /*
> * Calculate mapped_ratio under memory controller. This will be used in
> * vmscan.c for determining we have to reclaim mapped pages.
> @@ -532,6 +546,7 @@ static int mem_cgroup_charge_common(stru
> unsigned long flags;
> unsigned long nr_retries = MEM_CGROUP_RECLAIM_RETRIES;
> struct mem_cgroup_per_zone *mz;
> + enum res_state state;
>
> if (mem_cgroup_subsys.disabled)
> return 0;
> @@ -558,23 +573,23 @@ static int mem_cgroup_charge_common(stru
> mem = memcg;
> css_get(&memcg->css);
> }
> -
> - while (res_counter_charge(&mem->res, PAGE_SIZE) == RES_OVER_LIMIT) {
> +retry:
> + state = res_counter_charge(&mem->res, PAGE_SIZE);
> + if (state == RES_OVER_LIMIT) {
> + if (!(gfp_mask & __GFP_WAIT))
> + goto out;
> -
> + if (try_to_free_mem_cgroup_pages(mem, gfp_mask))
> - continue;
> -
> + goto retry;
> /*
> - * try_to_free_mem_cgroup_pages() might not give us a full
> - * picture of reclaim. Some pages are reclaimed and might be
> - * moved to swap cache or just unmapped from the cgroup.
> - * Check the limit again to see if the reclaim reduced the
> - * current usage of the cgroup before giving up
> + * try_to_free_mem_cgroup_pages() might not give us a
> + * full picture of reclaim. Some pages are reclaimed

```

```

> + * and might be moved to swap cache or just unmapped
> + * from the cgroup. Check the limit again to see if
> + * the reclaim reduced the current usage of the cgroup
> + * before giving up
> */
> if (res_counter_check_under_limit(&mem->res))
> - continue;
> + goto retry;
>
> if (!nr_retries--) {
>   mem_cgroup_out_of_memory(mem, gfp_mask);
> @@ -609,6 +624,9 @@ static int mem_cgroup_charge_common(stru
>   spin_unlock_irqrestore(&mz->lru_lock, flags);
>
>   unlock_page_cgroup(page);
> +
> + if (state > RES_BELOW_HIGH)
> +   mem_cgroup_schedule_reclaim(mem);

```

I really don't like state > RES_BELOW_HIGH sort of checks. Could we please abstract them into functions, like

```

if (mem_res_above_high_watermark(state))
....

> done:
> return 0;
> out:
> @@ -891,6 +909,74 @@ out:
>   css_put(&mem->css);
>   return ret;
> }
> +/*
> + * background reclaim daemon.
> + */
> +static int mem_cgroup_reclaim_daemon(void *data)
> +{
> + DEFINE_WAIT(wait);
> + struct mem_cgroup *mem = data;
> + int ret;
> +
> + css_get(&mem->css);
> + current->flags |= PF_SWAPWRITE;
> + /* we don't want to use cpu too much. */
> + set_user_nice(current, 0);

```

Shouldn't this (0) be a #define, what if we would like to degrade our nice value even further later?

```

> + set_freezable();
> +
> + while (!kthread_should_stop()) {
> +   prepare_to_wait(&mem->daemon.waitq, &wait, TASK_INTERRUPTIBLE);
> +   if (res_counter_state(&mem->res) == RES_BELOW_LOW) {
> +     if (!kthread_should_stop()) {
> +       schedule();
> +       try_to_freeze();

```

I am afraid, I fail to understand the code above.

```

> +   }
> +   finish_wait(&mem->daemon.waitq, &wait);
> +   continue;
> + }
> + finish_wait(&mem->daemon.waitq, &wait);
> + /*
> +  * memory resource controller doesn't see NUMA memory usage
> +  * balancing, because we cannot know what balancing is good.

```

~~~~~ typo

```

> +  * TODO: some annotation or heuristics to detect which node
> +  * we should start reclaim from.
> +  */
> + ret = try_to_free_mem_cgroup_pages(mem, GFP_HIGHUSER_MOVABLE);
> +
> + yield();

```

Why do we yeild here? Shouldn't our waitqueue handle the waiting?

```

> + }
> + css_put(&mem->css);
> + return 0;
> + }
> +
> +static int mem_cgroup_start_daemon(struct mem_cgroup *mem)
> +{
> + int ret = 0;
> + struct task_struct *thr;
> +
> + mutex_lock(&memcont_daemon_lock);
> + if (!mem->daemon.thread) {
> +   thr = kthread_run(mem_cgroup_reclaim_daemon, mem, "memcontd");
> +   if (IS_ERR(thr))
> +     ret = PTR_ERR(thr);
> +   else

```

```

> + mem->daemon.thread = thr;
> + }
> + mutex_unlock(&memcont_daemon_lock);
> + return ret;
> +}
> +
> +static void mem_cgroup_stop_daemon(struct mem_cgroup *mem)
> +{
> + mutex_lock(&memcont_daemon_lock);
> + if (mem->daemon.thread) {
> + kthread_stop(mem->daemon.thread);
> + mem->daemon.thread = NULL;
> + }
> + mutex_unlock(&memcont_daemon_lock);
> + return;
> +}
> +
>
> static int mem_cgroup_write_strategy(char *buf, unsigned long long *tmp)
> {
> @@ -915,6 +1001,19 @@ static ssize_t mem_cgroup_write(struct c
> struct file *file, const char __user *userbuf,
> size_t nbytes, loff_t *ppos)
> {
> + int ret;
> + /*
> + * start daemon can fail. But we should start daemon always
> + * when changes to HWMARK is succeeded. So, we start daemon before
> + * changes to HWMARK. We don't stop this daemon even if
> + * res_counter_write fails. To do that, we need ugly codes and
> + * it's not so big problem.
> + */
> + if (cft->private == RES_HWMARK) {
> + ret = mem_cgroup_start_daemon(mem_cgroup_from_cont(cont));
> + if (ret)
> + return ret;
> + }
> return res_counter_write(&mem_cgroup_from_cont(cont)->res,
> cft->private, userbuf, nbytes, ppos,
> mem_cgroup_write_strategy);
> @@ -1004,6 +1103,18 @@ static struct cftype mem_cgroup_files[]
> .read_u64 = mem_cgroup_read,
> },
> {
> + .name = "high_wmark_in_bytes",
> + .private = RES_HWMARK,
> + .write = mem_cgroup_write,
> + .read_u64 = mem_cgroup_read,

```

```

> + },
> + {
> + .name = "low_wmark_in_bytes",
> + .private = RES_LWMARK,
> + .write = mem_cgroup_write,
> + .read_u64 = mem_cgroup_read,
> + },
> + {
> .name = "force_empty",
> .trigger = mem_force_empty_write,
> },
> @@ -1087,7 +1198,8 @@ mem_cgroup_create(struct cgroup_subsys *
> return ERR_PTR(-ENOMEM);
> }
>
> - res_counter_init(&mem->res);
> + res_counter_init_wmark(&mem->res);
> + init_waitqueue_head(&mem->daemon.waitq);
>
> for_each_node_state(node, N_POSSIBLE)
> if (alloc_mem_cgroup_per_zone_info(mem, node))
> @@ -1106,6 +1218,7 @@ static void mem_cgroup_pre_destroy(struc
> struct cgroup *cont)
> {
> struct mem_cgroup *mem = mem_cgroup_from_cont(cont);
> + mem_cgroup_stop_daemon(mem);
> mem_cgroup_force_empty(mem);
> }
>
>

```

I failed to see code for the low watermark. Shouldn't we stop the thread once we go below the low watermark?

--

Warm Regards,  
Balbir Singh  
Linux Technology Center  
IBM, ISTL

---

Containers mailing list  
Containers@lists.linux-foundation.org  
<https://lists.linux-foundation.org/mailman/listinfo/containers>

---