
Subject: [Patch05/05]: Containers: Over the memory limit handler

Posted by [Rohit Seth](#) on Fri, 15 Sep 2006 01:43:28 GMT

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

This patch implements the handler for cases when container(s) go over their limit.

Signed-off-by: Rohit Seth <rohitseth@google.com>

mm/container_mm.c | 455

+++++

1 files changed, 455 insertions(+)

--- linux-2.6.18-rc6-mm2.org/mm/container_mm.c 1969-12-31 16:00:00.000000000 -0800

+++ linux-2.6.18-rc6-mm2.ctn/mm/container_mm.c 2006-09-14 16:50:25.000000000 -0700

@@ -0,0 +1,455 @@

+/*

+ * Copyright (c) 2006 Rohit Seth <rohitseth@google.com>

+ *

+ * Basic mm reclaimer for containers.

+ */

+

+#include <linux/mm.h>

+#include <linux/sched.h>

+#include <linux/container.h>

+#include <linux/rmap.h>

+#include <linux/mm_inline.h>

+#include <linux/writeback.h>

+#include <linux/swap.h>

+#include <linux/pagemap.h>

+#include <linux/pagevec.h>

+

+#if 0

+#define Cprintfk(x...) do { printfk(x); } while (0)

+#else

+#define Cprintfk(x...)

+#endif

+

+/*

+ * Count of task's vmas scanned for active page in one shot.

+ */

+#define MAX_VMA_COUNT 5

+

+/*

+ * Max number of active pages that get deactivated in each iteration

+ * of page_limit hit.

+ */

+#define DEACTIVATE_COUNT 100

```

+
+/*
+ * File size smaller than this number of pages is skipped in the
+ * first scan for inactivating pages. This is also used for scanning
+ * tasks with anon page count greater than this.
+ */
+#define PAGE_THRESHOLD_LIMIT 50
+
+/*
+ * Internal function that builds a list of vmas belonging to mm whose pages
+ * are going to be scanned to be deactivated. We only look for anonymous vmas
+ * It returns the number of vmas populated in vmas array.
+ */
+static int get_task_vmas(struct mm_struct *mm, struct vm_area_struct **vmas,
+ struct vm_area_struct *start_vma)
+{
+ struct vm_area_struct *tmp;
+ int i = 0;
+
+
+ if (start_vma == NULL)
+ tmp = mm->mmap;
+ else
+ tmp = start_vma->vm_next;
+
+ while ((i < MAX_VMA_COUNT) && (tmp != NULL)) {
+ vmas[i] = tmp;
+ if (!(tmp->vm_flags & (VM_IO | VM_RESERVED))) {
+ if (tmp->anon_vma != NULL)
+ i++;
+ }
+ tmp = tmp->vm_next;
+ }
+ return i;
+}
+
+/*
+ * This function iterates over vma's pages. It reinitializes the container
+ * pointer for a page if it belongs to ctn. It decrements the anon_page and
+ * active_page count for the container ctn.
+ * We only check for anonymous pages as the file pages get their container
+ * inheritance from mapping.
+ */
+static long anonpages_init(struct container_struct *ctn,
+ struct vm_area_struct *vma)
+{
+ struct page *page;
+ struct zone *z;

```

```

+ unsigned long addr, end;
+ long ret = 0, active = 0;
+
+ addr = vma->vm_start;
+ end = vma->vm_end;
+
+ while (addr < end) {
+   page = follow_page(vma, addr, 0);
+   if ((page) && (page->ctn == ctn)) {
+     z = page_zone(page);
+     spin_lock_irq(&z->lru_lock);
+     if (page->ctn == ctn) {
+       if (PageAnon(page)) {
+         page->ctn = NULL;
+         atomic_long_dec(&ctn->num_anon_pages);
+         ret++;
+         if (PageActive(page) && PageLRU(page))
+           active++;
+       }
+     }
+     spin_unlock_irq(&z->lru_lock);
+   }
+   addr += PAGE_SIZE;
+ }
+ atomic_long_sub(active, &ctn->num_active_pages);
+ return ret;
+}
+
+/*
+ * This function is called when a task is explicitly requested to be moved out
+ * of its container through /configs/containers/<container>/rmtask interface.
+ * We iterate over task's vmas and reinitialize the container pointer in
+ * every anon page mapping of this task.
+ */
+long anonpages_sub(struct task_struct *task, struct container_struct *ctn)
+{
+   struct vm_area_struct *vmas[MAX_VMA_COUNT];
+   struct vm_area_struct *start_vma = NULL;
+   struct mm_struct *mm;
+   long ret = 0;
+   int count, i;
+
+   mm = get_task_mm(task);
+   if (!mm)
+     return -1;
+   Cprintf("anonpages_sub scanning for task %d... ", task->pid);
+   /*
+    * No more new page faults beyond this point till the time

```

```

+ * we finish scanning.
+ */
+ down_write(&mm->mmap_sem);
+
+ /*
+ * Iterate over the vmas to deactivate pages.
+ */
+ while ((count = get_task_vmas(mm, vmas, start_vma)) > 0) {
+   i = 0;
+   while (i < count)
+     ret += anonpages_init(ctn, vmas[i++]);
+
+   if (count == MAX_VMA_COUNT)
+     /*
+      * Make get_task_vmas start from after the last
+      * scanned vma.
+      */
+     start_vma = vmas[MAX_VMA_COUNT-1];
+   else
+     break;
+ }
+ up_write(&mm->mmap_sem);
+ mmpu(mm);
+ Cprintf("returning %ld\n", ret);
+ return ret;
+}
+
+/*
+ * If the page is active then this function marks it inactive, clears the
+ * reference bit (so as to make it a better target for reclaim) and also
+ * puts it on inactive list. This function also decrements the zone's
+ * active number of pages.
+ */
+static int deactivate_container_page(struct page *page)
+{
+   struct zone *z;
+   int ret = 0, tmp;
+
+   if (!PageActive(page))
+     goto out;
+   if (!PageLRU(page))
+     goto out;
+
+   z = page_zone(page);
+   spin_lock_irq(&z->lru_lock);
+   if (PageActive(page) && PageLRU(page)) {
+     /*
+      * Not calling del_page_from_active_list so as to reduce

```



```

+ * We use the get_task_mm and mmpu to avoid handling of freed up mm.
+ */
+static int sync_container_anon_pages(struct container_struct *ctn)
+{
+ struct vm_area_struct *vmass[MAX_VMA_COUNT];
+ struct vm_area_struct *start_vma;
+ struct task_struct *tsk;
+ struct mm_struct *mm;
+ int i, nr = 0, count;
+ int vma_scanned = 0;
+ int small_task = 0;
+
+again:
+ list_for_each_entry(tsk, &ctn->tasks, ctn_task_list) {
+ start_vma = NULL;
+
+ if (tsk->flags & PF_EXITING)
+ continue;
+
+ mm = get_task_mm(tsk);
+ if (!mm)
+ continue;
+ if (!small_task)
+ if (get_mm_counter(mm, anon_rss) < PAGE_THRESHOLD_LIMIT)
+ goto next_task;
+ down_read(&mm->mmap_sem);
+ /*
+ * Iterate over the anonymous vmass to deactivate
+ * pages.
+ */
+ while ((count = get_task_vmass(mm, vmass, start_vma)) > 0) {
+ i = 0;
+
+ vma_scanned += count;
+ while ((i < count) && (nr < DEACTIVATE_COUNT))
+ nr += deactivate_pages(ctn, vmass[i++]);
+
+ if (nr >= DEACTIVATE_COUNT)
+ break;
+
+ if (count == MAX_VMA_COUNT)
+ /*
+ * Make get_task_vmass start from after the last
+ * scanned vma.
+ */
+ start_vma = vmass[MAX_VMA_COUNT-1];
+ else
+ break;

```

```

+ }
+ up_read(&mm->mmap_sem);
+next_task:
+ mmpu(m);
+
+ /*
+  * Break out of the loop if we have deactivated enough pages.
+  */
+ if ((atomic_long_read(&ctn->num_file_pages) +
+     atomic_long_read(&ctn->num_anon_pages)) <
+     (ctn->page_limit - nr))
+     goto out;
+ }
+ if (!small_task) {
+     small_task = 1;
+     goto again;
+ }
+out:
+ Cprintf("vma scanned %d  Anon returning: %d\n", vma_scanned, nr);
+ return nr;
+
+}
+endif /*CONFIG_SWAP */
+
+/*
+ * This function deactivates any of the active pages present in page
+ * array pl. count is the number of pages contained in the array.
+ */
+int deactivate_page_array(struct page **page_array, int count)
+{
+ int i = 0;
+ int ret = 0;
+ struct page *page;
+
+ while (i < count) {
+     page = page_array[i++];
+     ret += deactivate_container_page(page);
+     put_page(page);
+ }
+ return ret;
+}
+
+/*
+ * Find all the pages belonging to mapping and reinitialize
+ * its container pointer to point to ctn.
+ */
+long filepages_to_new_container(struct address_space *mapping,
+ struct container_struct *ctn)

```

```

+{
+ struct pagevec pvec;
+ long ret = 0;
+ int i;
+ pgoff_t next = 0;
+
+ pagevec_init(&pvec, 0);
+ while (pagevec_lookup(&pvec, mapping, next, PAGEVEC_SIZE)) {
+ ret += pagevec_count(&pvec);
+ for (i=0; i< pagevec_count(&pvec); i++) {
+ struct page *page = pvec.pages[i];
+ pgoff_t page_index = page->index;
+
+ page->ctn = ctn;
+ if (page_index > next)
+ next = page_index;
+ next++;
+ }
+ }
+ return ret;
+}
+
+/*
+ * This is the core of container file page reclaimer. It gets called
+ * whenever page_limit of a container is hit.
+ * This function traverses over the list of files belonging to this container
+ * and collects pages for each file till the time it can deactivate at least
+ * DEACTIVATE_COUNT pages.
+ * It is a two pass algorithm. First it looks for files that have larger than
+ * PAGE_THRESHOLD_LIMIT pages mapped. If we are not able to deactivate
+ * DEACTIVATE_COUNT number of pages then we do a second pass. This second
+ * pass scans smaller files only.
+ */
+static int sync_container_file_pages(struct container_struct *ctn)
+{
+ struct address_space *map;
+ struct pagevec pvec;
+ pgoff_t next = 0;
+ int small_files = 0;
+ int i;
+ int ret = 0;
+ int files_scanned = 0;
+ unsigned long invalidated = 0;
+
+again:
+ list_for_each_entry(map, &ctn->mappings, ctn_mapping_list) {
+ if (map->npages > 0)
+ continue;

```

```

+ if (!small_files) { /* Look for big files */
+ if (map->nrrpages < PAGE_THRESHOLD_LIMIT)
+ continue;
+ }
+ /*
+ * If the number of page cache pages itself is close to the
+ * limit then invalidate some of these pages here.
+ */
+ if (atomic_long_read(&ctn->num_file_pages) > ctn->page_limit)
+ invalidated += invalidate_inode_pages(map);
+
+ pagevec_init(&pvec, 0);
+ /*
+ * In the second iteration, deactivate more pages from each
+ * file.
+ */
+ while ((ret < (DEACTIVATE_COUNT * (small_files + 1))) &&
+ pagevec_lookup(&pvec, map, next, PAGEVEC_SIZE)) {
+ for (i=0; i< pagevec_count(&pvec); i++) {
+ struct page *page = pvec.pages[i];
+ pgoff_t page_index = page->index;
+
+ if (page_index > next)
+ next = page_index;
+ next++;
+ }
+ ret += deactivate_page_array(pvec.pages, pagevec_count(&pvec));
+ }
+ files_scanned++;
+ /*
+ * Break out of the loop if we have deactivated enough pages.
+ */
+ if ((atomic_long_read(&ctn->num_file_pages) +
+ atomic_long_read(&ctn->num_anon_pages)) <
+ (ctn->page_limit - ret))
+ goto out;
+ }
+
+ if ((atomic_long_read(&ctn->num_file_pages) +
+ atomic_long_read(&ctn->num_anon_pages)) >
+ (ctn->page_limit - ret) && !small_files) {
+ /* Iterate over smaller files also as we are not
+ * able to deactivate enough pages.
+ */
+ small_files = 1;
+ goto again;
+ }
+out:

```

```

+ atomic_long_sub(ret, &ctn->num_active_pages);
+ Cprintf("File returning: %d, invalidated %ld\n", ret, invalidated);
+ return ret;
+}
+
+/*
+ * This function is called when container's page limit is hit. In this case
+ * we move the pages (charged against this container) from active list
+ * to head of inactive list. So that if there is any memory pressure
+ * then these are the prime candidates for freeing. It first tries to inactivate
+ * file pages and then goes into inactivating anonymous pages.
+ * XXX: More smarts should be put here to get to the right pages
+ * faster and not start iterating over the same range again.
+ * Container's mutex is already held. This is in context of kcontainerd.
+ */
+void container_over_pagelimit(struct container_struct *ctn)
+{
+ int ret = 0;
+ long anon_pages, file_pages;
+ long tmp;
+ /*
+ * Grabbed the mutex so that container can not be freed and no new
+ * resource can be added while over the limit handler is working
+ * its way through.
+ */
+ file_pages = atomic_long_read(&ctn->num_file_pages);
+ anon_pages = atomic_long_read(&ctn->num_anon_pages);
+ tmp = file_pages + anon_pages;
+ if (tmp > ctn->page_limit) {
+ if (file_pages > (ctn->page_limit >> 4))
+ ret += sync_container_file_pages(ctn);
+ #ifdef CONFIG_SWAP
+ anon_pages = atomic_long_read(&ctn->num_anon_pages);
+ if ((total_swap_pages > 0) &&
+ (anon_pages > (ctn->page_limit >> 4)))
+ ret += sync_container_anon_pages(ctn);
+ #endif
+ }
+}

```
