Subject: [patch04/05]: Containers(V2)- Core container support
Posted by Rohit Seth on Wed, 20 Sep 2006 02:21:39 GMT
View Forum Message <> Reply to Message

This patch has the definitions and other core part of container support
implementing all the counters for different resources (like tasks, anon
memory etc.).

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

include/linux/container.h | 167 ++++++++++
mm/container.c | 658 ++
2 files changed, 825 insertions(+)

--- linux-2.6.18-rc6-mm2.org/include/linux/container.h 1969-12-31 16:00:00.000000000 -0800
+++ linux-2.6.18-rc6-mm2.ctn/include/linux/container.h 2006-09-19 12:58:28.000000000 -0700
@@ -0,0 +1,167 @@
+/*
+ * include/linux/container.h
+ * container header definitions for containers.
+ * Copyright (c) 2006 Rohit Seth <rohitseth@google.com>
+ */
+
+#ifndef _LINUX_CONTAINER_H
+#define _LINUX_CONTAINER_H
+
+#ifdef CONFIG_CONTAINERS
+
+#include <linux/kernel.h>
+#include <linux/types.h>
+#include <linux/mm.h>
+#include <linux/errno.h>
+#include <linux/nodemask.h>
+#include <linux/mutex.h>
+#include <linux/configfs.h>
+#include <linux/workqueue.h>
+
+#include <asm/system.h>
+#include <asm/page.h>
+#include <asm/atomic.h>
+
+struct task_struct;
+struct address_space;
+struct page;
+
+/*
+ * num_files is just a number indicating how many files are currently opened by
+ * task(s) belonging to this container.

```
+ */
+struct container_struct {
+ char *name; /* Container name */
+ int id; /* System wide container id */
+ int freeing; /* Used for marking a freeing container. */
+ unsigned long flags; /* See different bits below. */
+ atomic_t wait_on_mutex; /* Number of threads waiting to grab
+     * container mutex.  Used to make sure
+     * we don't free the container while there is
+     * someone waiting for mutex.
+     */
+
+ /* Accoutning fields. */
+ atomic_t num_tasks; /* Total number of threads. */
+ atomic_t num_files; /* See comment above. */
+ atomic_long_t num_anon_pages; /* Anonymous pages. */
+ atomic_long_t num_mapped_pages; /* File pages that are mapped. */
+ atomic_long_t num_file_pages; /* Pagecache pages. */
+ atomic_long_t num_active_pages; /* Pages on active list. */
+
+ /* Limits */
+ ssize_t page_limit; /* Max pages */
+ int task_limit; /* Max number of tasks */
+
+ /* Stats */
+ atomic_long_t page_limit_hits; /* Num times page limit is hit */
+ atomic_long_t task_limit_hits; /* Num times proc limit is hit. */
+
+ /* Mutex for resource list management. Also used while freeing. */
+ struct mutex mutex;
+
+ /* Various resource lists */
+ struct list_head tasks; /* List of tasks belonging to container */
+ struct list_head mappings; /* List of files belonging to container*/
+
+ u64 last_jiffies; /* Jiffy value when limits were hit last*/
+ struct work_struct work; /* Work structure for work_queues */
+ wait_queue_head_t mm_waitq; /* Wait queue when it goes over limit */
+};
+
+/*
+ * Container Flags here.
+ */
+#define CTN_OVER_MM_LIM 0 /* Container is over its memory limit. */
+
+extern int setup_container(struct container_struct *);
+extern int free_container(struct container_struct *);
+extern int container_add_task(struct task_struct *, struct task_struct *,
```

```
+  struct container_struct *);
+extern int container_add_file(struct address_space *, struct container_struct *);
+extern ssize_t set_container_page_limit(struct container_struct *, ssize_t);
+extern ssize_t set_container_task_limit(struct container_struct *, ssize_t);
+extern void container_remove_task(struct task_struct *, struct container_struct *);
+extern void container_remove_file(struct address_space *);
+extern void container_inc_page_count(struct page *);
+extern void container_dec_page_count(struct page *);
+extern void container_inc_filepage_count(struct address_space *, struct page *);
+extern void container_dec_filepage_count(struct page *);
+extern void container_inc_activepage_count(struct page *);
+extern void container_dec_activepage_count(struct page *);
+extern int freeing_container(struct container_struct  *);
+extern void container_over_pagelimit(struct container_struct *);
+extern void container_overlimit_handler(void *);
+extern long filepages_to_new_container(struct address_space *, struct container_struct *);
+extern long anonpages_sub(struct task_struct *, struct container_struct *);
+extern ssize_t container_show_tasks(struct container_struct *, char *);
+extern void writeback_container_file_pages(struct container_struct *);
+
+static inline void container_init_page_ptr(struct page *page,
+  struct task_struct *task)
+{
+ if (task == NULL)
+  page->ctn = NULL;
+ else
+  page->ctn = task->ctn;
+}
+
+static inline void container_init_task_ptr(struct task_struct *task)
+{
+ task->ctn = NULL;
+ INIT_LIST_HEAD(&task->ctn_task_list);
+}
+
+/*
+ * Following is for CONFIGFS
+ */
+struct simple_containerfs {
+ struct config_item item;
+ struct container_struct ctn;
+};
+
+struct simple_containerfs_attr {
+ struct configfs_attribute attr;
+ int idx; /* Indices defined below. */
+};
+
```

```
+/* Indices in configs directory.
+ */
+#define CONFIGFS_CTN_ATTR_ID    1
+#define CONFIGFS_CTN_ATTR_NUM_TASKS   2
+#define CONFIGFS_CTN_ATTR_NUM_FILES   3
+#define CONFIGFS_CTN_ATTR_NUM_ANON_PAGES  4
+#define CONFIGFS_CTN_ATTR_NUM_MAPPED_PAGES 5
+#define CONFIGFS_CTN_ATTR_NUM_FILE_PAGES  6
+#define CONFIGFS_CTN_ATTR_NUM_ACTIVE_PAGES  7
+#define CONFIGFS_CTN_ATTR_PAGE_LIMIT   8
+#define CONFIGFS_CTN_ATTR_TASK_LIMIT   9
+#define CONFIGFS_CTN_ATTR_PAGE_LIMIT_HITS 10
+#define CONFIGFS_CTN_ATTR_TASK_LIMIT_HITS 11
+#define CONFIGFS_CTN_ATTR_FREEING  12
+#define CONFIGFS_CTN_ATTR_ADD_TASK  13
+#define CONFIGFS_CTN_ATTR_RM_TASK  14
+#define CONFIGFS_CTN_ATTR_SHOW_TASKS  15
+#define CONFIGFS_CTN_ATTR_ADD_FILE  16
+#define CONFIGFS_CTN_ATTR_RM_FILE  17
+
+extern struct workqueue_struct *container_wq;
+
+#else /* CONFIG_CONTAINERS */
+
+#define container_add_task(t0, t1, container) 0
+#define container_add_file(mapping, container) do { } while(0)
+#define container_remove_task(task, container) do { } while(0)
+#define container_remove_file(address_space) do { } while(0)
+#define container_inc_page_count(page) do { } while(0)
+#define container_dec_page_count(page) do { } while(0)
+#define container_inc_filepage_count(address_space, page) do { } while(0)
+#define container_dec_filepage_count(page) do { } while(0)
+#define container_inc_activepage_count(page) do { } while(0)
+#define container_dec_activepage_count(page) do { } while(0)
+#define container_init_page_ptr(page, task) do { } while(0)
+#define container_init_task_ptr(task) do { } while(0)
+
+#endif  /* CONFIG_CONTAINERS */
+
+#endif /* LINUX_CONTAINER_H */
--- linux-2.6.18-rc6-mm2.org/mm/container.c 1969-12-31 16:00:00.000000000 -0800
+++ linux-2.6.18-rc6-mm2.ctn/mm/container.c 2006-09-19 13:54:38.000000000 -0700
@@ -0,0 +1,658 @@
+/*
+ * mm/coontainer.c
+ *
+ * Copyright (c) 2006 Rohit Seth <rohitseth@google.com>
+ */
```

```
+
+#include <linux/mm.h>
+#include <linux/rmap.h>
+#include <linux/container.h>
+
+/*
+ * If we hit some resource limit more often then MIN_JIFFY_INTERVAL
+ * then the wait time is doubled.
+ */
+#define MIN_JIFFY_INTERVAL 5
+
+/*
+ * Maximum number of contatiners allowed in a system.  Later
+ * make it configurable. Keep it multiple of BITS_PER_LONG
+ */
+#define MAX_CONTAINERS 512
+
+static int num_containers;
+/*
+ * Bit map array for container ids.
+ */
+DECLARE_BITMAP(ctn_bit_array, MAX_CONTAINERS);
+
+/*
+ * Protects container list updates.  This is only used
+ * when a container is getting created or destroyed.
+ */
+
+static DEFINE_SPINLOCK(container_lock);
+
+void wakeup_container_mm(struct container_struct *ctn);
+
+/*
+ * This function is called as part of creating a new container when a
+ * mkdir command is executed.  Container is already allocated as part
+ * of initialization in configfs directory.  Look at
+ * kernel/container_configfs.c
+ */
+int setup_container(struct container_struct *ctn)
+{
+ int idx;
+
+ spin_lock(&container_lock);
+
+ if (num_containers == (MAX_CONTAINERS -1)) {
+  idx = -EBUSY;
+  goto out;
+ }
```

```
+ idx = find_first_zero_bit(ctn_bit_array, MAX_CONTAINERS);
+ set_bit(idx, ctn_bit_array);
+ num_containers++;
+out:
+ spin_unlock(&container_lock);
+ ctn->id = idx;
+
+ return idx;
+
+}
+
+/*
+ * This function is reached when user executes
+ * echo <pid> > /configfs/container/<container>/addtask command.  It is also
+ * called at fork time.  If a task is  not already a part of another
+ * container, it is allowed to move into this container.
+ * The two fields in task_struct,
+ * 1- ctn: is initialized to point to this container.
+ * 2- ctn_list: task is added to the head of tasks list for the container.
+ * XXX: Not adding any existing pages belonging to task to this container.
+ * If we are getting called from fork path then parent is non-NULL task
+ * pointer and ctn is NULL.
+ * If we are getting called from configfs interface then parent is
+ * NULL but ctn will point to the non-NULL container.
+ */
+int container_add_task(struct task_struct *task, struct task_struct *parent,
+  struct container_struct *container)
+{
+ int ret = 0;
+ struct container_struct *ctn = container;
+
+ /* First take care of processes running out side of container.
+  */
+ if (parent && (parent->ctn == NULL))
+  return 0;
+
+ if (ctn == NULL) {
+  /*
+   * We do this here so that parent's container pointer is not
+   * replaced underneath us.
+   */
+  task_lock(parent);
+  if (parent->ctn) {
+   ctn = parent->ctn;
+   atomic_inc(&ctn->wait_on_mutex);
+  }
+  task_unlock(parent);
+  /*
```

```
+   * If some one else removed parent from a container then we
+   * just return as in this case task will not be added to any
+   * container.
+   */
+  if (ctn == NULL)
+   return 0;
+ } else
+  atomic_inc(&ctn->wait_on_mutex);
+
+ /*
+  * Easy check first outside of mutex if the container is getting freed.
+  */
+ if (ctn->freeing) {
+  ret = -ENOENT;
+  goto out_no_mutex;
+ }
+
+ mutex_lock(&ctn->mutex);
+ /*
+  * First check if the container is not already marked for freeing.
+  */
+ if (ctn->freeing) {
+  ret = -ENOENT;
+  goto out_no_lock;
+ }
+
+ task_lock(task);
+
+ if (task->ctn) {
+  /*
+   * Can not move a process from one container to another
+   */
+  ret = -EINVAL;
+  goto out;
+ }
+
+ if (atomic_read(&ctn->num_tasks) >= ctn->task_limit) {
+  ret =  -ENOSPC;
+  goto out;
+ }
+ atomic_inc(&ctn->num_tasks);
+ task->ctn = ctn;
+
+ /*
+  * Add the process to link list of tasks belonging to this container
+  */
+ list_add(&task->ctn_task_list, &ctn->tasks);
+
```

```
+out:
+ task_unlock(task);
+out_no_lock:
+ mutex_unlock(&ctn->mutex);
+out_no_mutex:
+ atomic_dec(&ctn->wait_on_mutex);
+
+ return ret;
+}
+
+/*
+ * This function checks if the count is greater than container's page_limit.
+ * If so then it wakes up memory controller.  If the container is gone
+ * way over its limit (currently set at about 12% of page_limit) then we
+ * schedule the current process out.  And if the limit is happening too often
+ * then increase the timeout.
+ */
+int check_page_limit(struct container_struct *ctn, long count)
+{
+ int ret = 0;
+ int limit = ctn->page_limit;
+
+ /*
+  * Don't wake the over the limit memory handler if there are not
+  * enough pages on the active list.  This will allow the containers
+  * to have more inactive pages.
+  */
+ if ((count > limit) &&
+   (atomic_long_read(&ctn->num_active_pages) > limit)) {
+
+  wakeup_container_mm(ctn);
+  ret = 1;
+  /* If we are way over the limit then schedule out.
+   */
+  if (count > (limit + (limit >> 3))) {
+   int wait = HZ;
+   u64 temp;
+
+   /* No lock here and only one process may wait longer in case
+    * multiple processes are banging too hard.
+    */
+   temp = ctn->last_jiffies;
+   ctn->last_jiffies = get_jiffies_64();
+   if ((ctn->last_jiffies - temp) < MIN_JIFFY_INTERVAL)
+    wait *= 2;
+   wait_event_interruptible_timeout(ctn->mm_waitq,
+     (ctn->flags & CTN_OVER_MM_LIM) == 0, wait);
+  }
```

```
+ }
+
+ return ret;
+}
+
+/*
+ * This function is called whenever an anonymous or file backed page is getting
+ * mapped for the first time.  We first check if the
+ * page belongs to any container (which is already set at the time when page's
+ * association was made with either anon_vma or mapping).
+ *
+ * This function updates number of anonymous pages OR mapped pages belonging
+ * to this container.  If the total memory (anon + file backed) usage has
+ * exceeded the page_limit for the container, we put this container on the work
+ * queue for the memory controller.
+ *
+ * We also increment the number of times page limit is hit for this
+ * container.
+ */
+void container_inc_page_count(struct page *page)
+{
+ struct container_struct *ctn = page->ctn;
+ long tmp;
+
+ if (ctn == NULL)
+  return;
+
+ if (PageAnon(page))
+  atomic_long_inc(&ctn->num_anon_pages);
+ else
+  atomic_long_inc(&ctn->num_mapped_pages);
+
+ /*
+  * We check the limits against the total pages present in page cache
+  * and not only the ones that are mapped.
+  */
+ tmp = atomic_long_read(&ctn->num_anon_pages) +
+  atomic_long_read(&ctn->num_file_pages);
+ if (check_page_limit(ctn, tmp))
+  atomic_long_inc(&ctn->page_limit_hits);
+}
+
+/*
+ * This function is called whenever user (both anonymous or file backed) page
+ * is getting freed.  First we check if this page belongs to any container or
+ * not.  It decrements the number of anonymous OR File backed pages belonging
+ * to this container.
+ * It is not possible for container to have empty task list but non-zero
```

```
+ * num_anon_pages or num_mapped_pages count.
+ */
+void container_dec_page_count(struct page  *page)
+{
+ struct container_struct *ctn = page->ctn;
+
+ if (ctn == NULL)
+  return;
+ if (PageAnon(page)) {
+  if (atomic_long_read(&ctn->num_anon_pages) > 0)
+   atomic_long_dec(&ctn->num_anon_pages);
+  else
+   printk(KERN_WARNING"Container: Wrong Anon page count\n");
+ } else {
+  if (atomic_long_read(&ctn->num_mapped_pages) > 0)
+   atomic_long_dec(&ctn->num_mapped_pages);
+  else
+   printk(KERN_WARNING"Container: Wrong Mapped page count\n");
+ }
+}
+
+/*
+ * This function is called whenever a pagecache page is allocated to
+ * address space.  This function  updates
+ * 1- the per page container field and
+ * 2- number of pagecache(num_file_pages) pages belonging to this container.
+ * If the total memory (anon + file backed) usage has exceeded the
+ * page_limit for the container, we put this container on the work
+ * queue for the memory controller.
+ * We also increment the number of times page limit is hit for this
+ * container.
+ */
+void container_inc_filepage_count(struct address_space *map, struct page *page)
+{
+ struct container_struct *ctn = map->ctn;
+ long tmp;
+
+ if (ctn == NULL)
+  return;
+ page->ctn = ctn;
+ atomic_long_inc(&ctn->num_file_pages);
+ tmp = atomic_long_read(&ctn->num_anon_pages) + atomic_long_read(&ctn->num_file_pages);
+ /*
+  * check if the number of page cache pages is already close
+  * to the limit.  If that is the case then we initiate the writeback
+  * on some of these pages.
+  */
+ if (atomic_long_read(&ctn->num_file_pages) >
```

```
+   (ctn->page_limit - (ctn->page_limit >> 3))) {
+  mutex_lock(&ctn->mutex);
+  writeback_container_file_pages(ctn);
+  mutex_unlock(&ctn->mutex);
+ }
+}
+
+/*
+ * This function is called whenever a page is getting removed from
+ * pagecache.  This function decrements the number of pagecache pages
+ * belonging to this container.
+ */
+void container_dec_filepage_count(struct page *page)
+{
+ struct container_struct *ctn = page->ctn;
+
+ if (ctn == NULL)
+  return;
+ if (atomic_long_read(&ctn->num_file_pages) > 0)
+  atomic_long_dec(&ctn->num_file_pages);
+}
+
+/*
+ * This function is called whenever a page is getting added to
+ * zone's active list.  This information is used
+ * by mm controller to see if more pages need to be put
+ * on inactive list. This function updates increments the number of active
+ * pages belonging to * this container.
+ * zone's lock is already held, so do absolutely minimal and return fast.
+ */
+void container_inc_activepage_count(struct page *page)
+{
+ struct container_struct *ctn = page->ctn;
+
+ if (ctn == NULL)
+  return;
+ atomic_long_inc(&ctn->num_active_pages);
+ if (atomic_long_read(&ctn->num_active_pages) > ctn->page_limit)
+  wakeup_container_mm(ctn);
+}
+
+/*
+ * This function is called whenever a page is getting removed from
+ * zone's active list.
+ */
+void container_dec_activepage_count(struct page *page)
+{
+ struct container_struct *ctn = page->ctn;
```

```
+
+ if (ctn)
+  atomic_long_dec(&ctn->num_active_pages);
+}
+
+/*
+ * This function is called:
+ * 1- When a file is getting assigned to a different container.
+ * 2- When file's inode is getting removed.
+ * It grabs the container's mutex and updates the mapping list of the container.
+ */
+void container_remove_file(struct address_space *map)
+{
+ struct container_struct *ctn = map->ctn;
+
+ if (ctn == NULL)
+  return;
+ mutex_lock(&ctn->mutex);
+ atomic_dec(&ctn->num_files);
+ list_del(&map->ctn_mapping_list);
+ map->ctn = NULL;
+ mutex_unlock(&ctn->mutex);
+}
+
+/*
+ * This function is reached when user executes
+ * echo <file_name> > /configfs/container/test_container/addfile command
+ * (not yet implemented).  It is also called when a new inode is created as a
+ * result of some filesystem related operation.
+ * If the file already belongs to another container (when configfs interface
+ * works), then it is removed from that container first. All
+ * existing pages are migrated to new container.
+ * The two fields in address_space:
+ * 1- ctn: is initialized to point to this container.
+ * 2- ctn_mapping_list: address_space is added to the head of address_space
+ * list for the container.
+ */
+int container_add_file(struct address_space *map, struct container_struct *ctn)
+{
+ int ret = 0;
+ long count = 0;
+
+ /*
+  * If it is due to new inode allocation.
+  */
+ if (ctn == NULL)
+  ctn = current->ctn;
+ if (map->ctn == ctn)
```

```
+  return ret;
+ count = map->nrpages;
+ if (map->ctn)  {/* Already belonging to a container. */
+  container_remove_file(map);
+  /* Now initialize the per page pointer to point to
+   * new container.
+   */
+  count = filepages_to_new_container(map, ctn);
+  atomic_long_sub(count, &map->ctn->num_file_pages);
+ }
+
+ mutex_lock(&ctn->mutex);
+ if (map->ctn != ctn) {
+  INIT_LIST_HEAD(&map->ctn_mapping_list);
+  atomic_inc(&ctn->num_files);
+  map->ctn = ctn;
+  atomic_long_add(count, &ctn->num_file_pages);
+  list_add(&map->ctn_mapping_list, &ctn->mappings);
+ }
+ mutex_unlock(&ctn->mutex);
+
+ return ret;
+}
+
+/*
+ * This function is reached when user executes
+ * echo number > /configfs/container/test_container/page_limit command. It
+ * updates the page_limit field of container.  If the current consumption of
+ * memory is above the new page_limit then memory controller is called
+ * in the same way as if the original page limit is hit.  It is possible that
+ * controller may not be able to bring the consumption below the limits
+ * immediately.
+ */
+ssize_t set_container_page_limit(struct container_struct *ctn, ssize_t count)
+{
+
+ ctn->page_limit =  count;
+ if (count < (atomic_long_read(&ctn->num_anon_pages)
+    + atomic_long_read(&ctn->num_file_pages))) {
+  wakeup_container_mm(ctn);
+  atomic_long_inc(&ctn->page_limit_hits);
+ }
+ return count;
+}
+
+/*
+ * This function is reached when user executes
+ * echo number > /configfs/container/test_container/task_limit command. It
```

```
+ * updates the task_limit field of container.  No effort is made to see if
+ * the current consumption is above the new limits. Though we do update the
+ * number of task limit_hits.
+ * XXX: Adding killing of tasks.
+ */
+ssize_t set_container_task_limit(struct container_struct *ctn, ssize_t count)
+{
+ ctn->task_limit = (int)count;
+ if (count < atomic_read(&ctn->num_tasks)) {
+  atomic_long_inc(&ctn->task_limit_hits);
+ }
+ return count;
+}
+
+/*
+ * This function can be called in two ways:
+ * 1- At the time of task's exit (PF_EXITING is set)
+ * 2- Through /configfs/containers/<container_name>/rmtask interface
+ *
+ * We grab the container mutex and task lock to
+ * 1- initialize the task's container pointer
+ * 2- If PF_EXITIING is not set then move the anon pages belonging to
+ * this container out (uncharge this container).
+ */
+void container_remove_task(struct task_struct *task,
+  struct container_struct *ctn)
+{
+ struct container_struct *temp;
+ int count;
+
+
+ /*
+  * This quick check for cases where task does not belong to any
+  * container.  It is okay for this check to race with task->ctn
+  * modifications that might be happening under task lock.
+  */
+ if (task->ctn == NULL)
+  return;
+ /*
+  * This time we are acquiring the task lock only.  We acquire the task
+  * lock and check if there is any other parallel remove_task operation
+  * going on at this time on task's container.  We want this operation
+  * to be serial as it
+  * could be racing with free_container operation.  If we don't avoid
+  * this race then it is possible that we might end up holding a
+  * container mutex here which has already been freed.
+  * ...Cost of providing asynchronous task removal from containers.
+  */
```

```
+ task_lock(task);
+ temp = task->ctn;
+ if (temp == NULL)
+  /*
+   * Someone else won the race and removed the task from
+   * its container.
+   */
+  goto out;
+
+ /*
+  * ctn will be NULL for the exit cases only.  We inc the
+  * wait_on_mutex only if it is in exit path (and task belongs
+  * to some container) OR if the task is contained in ctn.
+  */
+ if ((ctn == NULL) || (temp == ctn))
+  atomic_inc(&temp->wait_on_mutex);
+ else
+  /*
+   * Task does not belong to ctn any more.
+   */
+  goto out;
+ task_unlock(task);
+
+ mutex_lock(&temp->mutex);
+ task_lock(task);
+ if (task->ctn != temp) {
+  mutex_unlock(&temp->mutex);
+  atomic_dec(&temp->wait_on_mutex);
+  goto out;
+ }
+ task->ctn = NULL;
+ list_del(&task->ctn_task_list);
+ task_unlock(task);
+ atomic_dec(&temp->num_tasks);
+ mutex_unlock(&temp->mutex);
+ atomic_dec(&temp->wait_on_mutex);
+
+ if (!(task->flags & PF_EXITING))
+  /* Don't need this count at this point.
+   */
+  count = anonpages_sub(task, temp);
+ return;
+out:
+ task_unlock(task);
+ return;
+}
+
+void container_remove_tasks(struct container_struct *ctn)
```

```
+{
+ struct task_struct *tsk;
+
+ list_for_each_entry(tsk, &ctn->tasks, ctn_task_list)
+  container_remove_task(tsk, ctn);
+}
+
+/*
+ * Following function is used to show pid of each task belonging to ctn.
+ */
+ssize_t container_show_tasks(struct container_struct *ctn, char *buf)
+{
+ struct task_struct *tsk;
+ ssize_t ret = 0;
+ int i = 0;
+
+ if (ctn->freeing)
+  return 0;
+ atomic_inc(&ctn->wait_on_mutex);
+ mutex_lock(&ctn->mutex);
+ list_for_each_entry(tsk, &ctn->tasks, ctn_task_list) {
+  ret += sprintf(buf+ret, "%d ", tsk->pid);
+  if (++i == 12) {
+   i = 0;
+   ret += sprintf(buf+ret, "\n");
+  }
+ }
+ mutex_unlock(&ctn->mutex);
+ atomic_dec(&ctn->wait_on_mutex);
+ if (i != 0)
+  ret += sprintf(buf+ret, "\n");
+ return ret;
+}
+
+void container_remove_files(struct container_struct *ctn)
+{
+ struct address_space *mapping;
+ long count;
+
+ list_for_each_entry(mapping, &ctn->mappings, ctn_mapping_list) {
+  container_remove_file(mapping);
+  count = filepages_to_new_container(mapping, NULL);
+  atomic_long_sub(count, &ctn->num_file_pages);
+ }
+}
+
+/*
+ * This function is called as part of deleting a container when a
```

```
+ * rmdir command is executed in configfs.  Container is already
+ * marked getting freed (ctn->freeing).  Look at  kernel/container_configfs.c
+ * Once we get in here, no new object can be added to this container.
+ * Actual container also gets freed at the same place in configfs support.
+ * We will move any existing resources like task out of container and wait
+ * for all the pending operations to complete before returning.
+ */
+int free_container(struct container_struct *ctn)
+{
+ int idx;
+ int ret;
+
+ idx = ctn->id;
+
+again:
+ if (atomic_read(&ctn->num_tasks))
+  container_remove_tasks(ctn);
+ if (atomic_read(&ctn->num_files))
+  container_remove_files(ctn);
+ mutex_lock(&ctn->mutex);
+
+ /*
+  * The check for num_anon_pages is there because when a task
+  * is removed from container using rmtask attribute, we reinitialize
+  * page's container pointer outside the container mutex.
+  */
+ if (atomic_read(&ctn->num_tasks) || ctn->flags ||
+   (atomic_read(&ctn->wait_on_mutex)) ||
+   atomic_read(&ctn->num_files) ||
+   atomic_long_read(&ctn->num_anon_pages)) {
+  mutex_unlock(&ctn->mutex);
+  schedule();
+  goto again;
+ }
+ spin_lock(&container_lock);
+ num_containers--;
+ clear_bit(idx, ctn_bit_array);
+ spin_unlock(&container_lock);
+
+ ctn->id = -1;
+ ret = 0;
+ mutex_unlock(&ctn->mutex);
+
+ return ret;
+}
+
+/*
+ * This function wakes up mm controller (if it is not already active).  It
```

```
+ * sets container's CTN_OVER_MM_LIM flag to indicate this container has
+ * hit the page limit and needs controller's action.
+ */
+void wakeup_container_mm(struct container_struct *ctn)
+{
+ if (test_and_set_bit(CTN_OVER_MM_LIM, &ctn->flags))
+   return;
+ queue_work(container_wq, &ctn->work);
+}
+
+/*
+ * Container over the limit controller/handler...implemented as kernel thread.
+ * It picks up its work from workqueue and calls the core controller routine
+ * container_over_pagelimit (in file  mm/container_mm.c when page_limit is hit).
+ */
+
+void container_overlimit_handler(void *p)
+{
+ struct container_struct *ctn = (struct container_struct *)p;
+
+ if (test_bit(CTN_OVER_MM_LIM, &ctn->flags)) { /* Mem handler */
+  mutex_lock(&ctn->mutex);
+  container_over_pagelimit(ctn);
+  clear_bit(CTN_OVER_MM_LIM, &ctn->flags);
+  if (ctn->page_limit < atomic_read(&ctn->num_active_pages))
+   wake_up_interruptible_all(&ctn->mm_waitq);
+  mutex_unlock(&ctn->mutex);
+ }
+}
```