Subject: [patch03/05]: Containers(V2)- Container initialization and configfs interface
Posted by Rohit Seth on Wed, 20 Sep 2006 02:21:10 GMT
View Forum Message <> Reply to Message

This patch contains containers interface with configfs. This patch
defines config group for containers underneath which different
containers can be created. This patch also contains initialization code
for creating work queue.

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

kernel/container_configfs.c | 440
++
1 files changed, 440 insertions(+)

--- linux-2.6.18-rc6-mm2.org/kernel/container_configfs.c 1969-12-31 16:00:00.000000000 -0800
+++ linux-2.6.18-rc6-mm2.ctn/kernel/container_configfs.c 2006-09-19 11:34:16.000000000 -0700
@@ -0,0 +1,440 @@
+/*
+ * Copyright (c) 2006 Rohit Seth <rohitseth@google.com>
+ *
+ * Container initialization code and configfs registration code.
+ */
+
+#include <linux/init.h>
+#include <linux/module.h>
+#include <linux/slab.h>
+#include <linux/container.h>
+#include <linux/configfs.h>
+#include <linux/workqueue.h>
+
+/*
+ * Per processor worker thread for handling over the limits scenarios.
+ */
+struct workqueue_struct *container_wq;
+
+/*
+ * Default limit for physical memory for a newly created container. Can
+ * be changed through /configfs/container/<container_name>/page_limit
+ */
+#define DEFAULT_PAGE_LIMIT ((900*1024*1024) >> PAGE_SHIFT)
+
+/*
+ * Default limit for number of tasks that can be created with in a container. Can
+ * be changed through /configfs/container/<container_name>/task_limit
+ */
+#define DEFAULT_TASK_LIMIT 100
+

```
+/*
+ * Following attributes are defined as the interface mechanism between configfs
+ * and containers.
+ * ca_name is the name as it gets shown in configfs
+ * ca_mode is the mode of that attribute
+ * idx is index of the attribute in the container.  This is used to find out
+ * what specific operation is requested.
+ */
+static struct simple_containerfs_attr simple_containerfs_attr_id = {
+ .attr = { .ca_owner = THIS_MODULE, .ca_name = "id", .ca_mode = S_IRUGO },
+ .idx = CONFIGFS_CTN_ATTR_ID,
+};
+
+static struct simple_containerfs_attr simple_containerfs_attr_num_tasks = {
+ .attr = { .ca_owner = THIS_MODULE, .ca_name = "num_tasks", .ca_mode = S_IRUGO },
+ .idx = CONFIGFS_CTN_ATTR_NUM_TASKS,
+};
+
+static struct simple_containerfs_attr simple_containerfs_attr_num_files = {
+ .attr = { .ca_owner = THIS_MODULE, .ca_name = "num_files", .ca_mode = S_IRUGO },
+ .idx = CONFIGFS_CTN_ATTR_NUM_FILES,
+};
+
+static struct simple_containerfs_attr simple_containerfs_attr_num_anon_pages = {
+ .attr = { .ca_owner = THIS_MODULE, .ca_name = "num_anon_pages", .ca_mode = S_IRUGO
},
+ .idx = CONFIGFS_CTN_ATTR_NUM_ANON_PAGES,
+};
+
+static struct simple_containerfs_attr simple_containerfs_attr_num_mapped_pages = {
+ .attr = { .ca_owner = THIS_MODULE, .ca_name = "num_mapped_pages", .ca_mode =
S_IRUGO },
+ .idx = CONFIGFS_CTN_ATTR_NUM_MAPPED_PAGES,
+};
+
+static struct simple_containerfs_attr simple_containerfs_attr_num_file_pages = {
+ .attr = { .ca_owner = THIS_MODULE, .ca_name = "num_file_pages", .ca_mode = S_IRUGO },
+ .idx = CONFIGFS_CTN_ATTR_NUM_FILE_PAGES,
+};
+
+static struct simple_containerfs_attr simple_containerfs_attr_num_active_pages = {
+ .attr = { .ca_owner = THIS_MODULE, .ca_name = "num_active_pages", .ca_mode = S_IRUGO
},
+ .idx = CONFIGFS_CTN_ATTR_NUM_ACTIVE_PAGES,
+};
+
+static struct simple_containerfs_attr simple_containerfs_attr_page_limit = {
+ .attr = { .ca_owner = THIS_MODULE, .ca_name = "page_limit", .ca_mode = S_IRUGO |
```

```
S_IWUSR },
+ .idx = CONFIGFS_CTN_ATTR_PAGE_LIMIT,
+};
+
+static struct simple_containerfs_attr simple_containerfs_attr_task_limit = {
+ .attr = { .ca_owner = THIS_MODULE, .ca_name = "task_limit", .ca_mode = S_IRUGO |
S_IWUSR },
+ .idx = CONFIGFS_CTN_ATTR_TASK_LIMIT,
+};
+
+static struct simple_containerfs_attr simple_containerfs_attr_page_limit_hits = {
+ .attr = { .ca_owner = THIS_MODULE, .ca_name = "page_limit_hits", .ca_mode = S_IRUGO },
+ .idx = CONFIGFS_CTN_ATTR_PAGE_LIMIT_HITS,
+};
+
+static struct simple_containerfs_attr simple_containerfs_attr_task_limit_hits = {
+ .attr = { .ca_owner = THIS_MODULE, .ca_name = "task_limit_hits", .ca_mode = S_IRUGO },
+ .idx = CONFIGFS_CTN_ATTR_TASK_LIMIT_HITS,
+};
+
+static struct simple_containerfs_attr simple_containerfs_attr_freeing = {
+ .attr = { .ca_owner = THIS_MODULE, .ca_name = "freeing", .ca_mode = S_IRUGO },
+ .idx = CONFIGFS_CTN_ATTR_FREEING,
+};
+
+static struct simple_containerfs_attr simple_containerfs_attr_addtask = {
+ .attr = { .ca_owner = THIS_MODULE, .ca_name = "addtask", .ca_mode = S_IRUGO |
S_IWUSR },
+ .idx = CONFIGFS_CTN_ATTR_ADD_TASK,
+};
+
+static struct simple_containerfs_attr simple_containerfs_attr_rmtask = {
+ .attr = { .ca_owner = THIS_MODULE, .ca_name = "rmtask", .ca_mode = S_IRUGO | S_IWUSR
},
+ .idx = CONFIGFS_CTN_ATTR_RM_TASK,
+};
+
+static struct simple_containerfs_attr simple_containerfs_attr_showtasks = {
+ .attr = { .ca_owner = THIS_MODULE, .ca_name = "showtasks", .ca_mode = S_IRUGO },
+ .idx = CONFIGFS_CTN_ATTR_SHOW_TASKS,
+};
+
+static struct simple_containerfs_attr simple_containerfs_attr_addfile = {
+ .attr = { .ca_owner = THIS_MODULE, .ca_name = "addfile", .ca_mode = S_IRUGO | S_IWUSR
},
+ .idx = CONFIGFS_CTN_ATTR_ADD_FILE,
+};
+
```

```
+static struct simple_containerfs_attr simple_containerfs_attr_rmfile = {
+ .attr = { .ca_owner = THIS_MODULE, .ca_name = "rmfile", .ca_mode = S_IRUGO | S_IWUSR },
+ .idx = CONFIGFS_CTN_ATTR_RM_FILE,
+};
+
+/*
+ * Array of all the attributes defined for containers. Used at the time when
+ * container is created.
+ */
+static struct configfs_attribute *simple_containerfs_attrs[] = {
+ &simple_containerfs_attr_id.attr,
+ &simple_containerfs_attr_num_tasks.attr,
+ &simple_containerfs_attr_num_files.attr,
+ &simple_containerfs_attr_num_anon_pages.attr,
+ &simple_containerfs_attr_num_mapped_pages.attr,
+ &simple_containerfs_attr_num_file_pages.attr,
+ &simple_containerfs_attr_num_active_pages.attr,
+ &simple_containerfs_attr_page_limit.attr,
+ &simple_containerfs_attr_task_limit.attr,
+ &simple_containerfs_attr_page_limit_hits.attr,
+ &simple_containerfs_attr_task_limit_hits.attr,
+ &simple_containerfs_attr_freeing.attr,
+ &simple_containerfs_attr_addtask.attr,
+ &simple_containerfs_attr_rmtask.attr,
+ &simple_containerfs_attr_showtasks.attr,
+ &simple_containerfs_attr_addfile.attr,
+ &simple_containerfs_attr_rmfile.attr,
+ NULL,
+};
+
+static inline struct simple_containerfs *to_simple_containerfs(struct config_item *item)
+{
+ return item ? container_of(item, struct simple_containerfs, item) : NULL;
+}
+
+/*
+ * simple_containerfs_attr_show operation is executed when ever there is a
+ * read operation on any of container's attribute.
+ */
+static ssize_t simple_containerfs_attr_show(struct config_item *item,
+  struct configfs_attribute *attr,
+  char *buf)
+{
+ struct simple_containerfs *sc = to_simple_containerfs(item);
+ struct simple_containerfs_attr  *ctfs_attr =
+  container_of(attr, struct simple_containerfs_attr, attr);
+ ssize_t tmp;
+ int copied = 0;
```

```
+
+ /* Attributes's index tells us what operation is requested. */
+ switch (ctfs_attr->idx) {
+ case CONFIGFS_CTN_ATTR_ID:
+  tmp = sc->ctn.id;
+  break;
+ case CONFIGFS_CTN_ATTR_NUM_TASKS:
+  tmp = (ssize_t)atomic_read(&sc->ctn.num_tasks);
+  break;
+ case CONFIGFS_CTN_ATTR_NUM_FILES:
+  tmp = (ssize_t)atomic_read(&sc->ctn.num_files);
+  break;
+ case CONFIGFS_CTN_ATTR_NUM_ANON_PAGES:
+  tmp = atomic_long_read(&sc->ctn.num_anon_pages);
+  break;
+ case CONFIGFS_CTN_ATTR_NUM_MAPPED_PAGES:
+  tmp = atomic_long_read(&sc->ctn.num_mapped_pages);
+  break;
+ case CONFIGFS_CTN_ATTR_NUM_FILE_PAGES:
+  tmp = atomic_long_read(&sc->ctn.num_file_pages);
+  break;
+ case CONFIGFS_CTN_ATTR_NUM_ACTIVE_PAGES:
+  tmp = atomic_long_read(&sc->ctn.num_active_pages);
+  break;
+ case CONFIGFS_CTN_ATTR_PAGE_LIMIT:
+  tmp = sc->ctn.page_limit;
+  break;
+ case CONFIGFS_CTN_ATTR_TASK_LIMIT:
+  tmp = sc->ctn.task_limit;
+  break;
+ case CONFIGFS_CTN_ATTR_PAGE_LIMIT_HITS:
+  tmp = atomic_long_read(&sc->ctn.page_limit_hits);
+  break;
+ case CONFIGFS_CTN_ATTR_TASK_LIMIT_HITS:
+  tmp = atomic_long_read(&sc->ctn.task_limit_hits);
+  break;
+ case CONFIGFS_CTN_ATTR_FREEING:
+  tmp = sc->ctn.freeing;
+  break;
+ case CONFIGFS_CTN_ATTR_SHOW_TASKS:
+  copied = 1;
+  tmp = container_show_tasks(&sc->ctn, buf);
+  break;
+ default:
+  tmp = -1;
+ }
+
+ if (!copied)
```

```
+  tmp = sprintf(buf, "%ld\n", tmp);
+ return tmp;
+
+}
+
+static ssize_t simple_containerfs_attr_store(struct config_item *item,
+           struct configfs_attribute *attr,
+           const char *buf, size_t count)
+{
+ struct simple_containerfs *sc = to_simple_containerfs(item);
+ struct simple_containerfs_attr  *ctfs_attr =
+  container_of(attr, struct simple_containerfs_attr, attr);
+ ssize_t tmp = 0;
+ char *p = (char *)buf;
+
+ /*
+  * For now it is only a simple operation.  Expected input is
+  * integer for the attributes that are less than or equal to
+  * CONFIGFS_CTN_ATTR_RM_TASK defined in include/linux/container.h
+  * But update this code later as different types are expected.
+  */
+ if (ctfs_attr->idx <= CONFIGFS_CTN_ATTR_RM_TASK) {
+  tmp = simple_strtoul(p, &p, 10);
+  if (!p || (*p && (*p != '\n')))
+   return -EINVAL;
+ }
+
+ if (tmp > INT_MAX)
+  return -ERANGE;
+
+ switch (ctfs_attr->idx) {
+ case CONFIGFS_CTN_ATTR_PAGE_LIMIT:
+  tmp = set_container_page_limit(&sc->ctn, tmp);
+  break;
+ case CONFIGFS_CTN_ATTR_TASK_LIMIT:
+  tmp = set_container_task_limit(&sc->ctn, tmp);
+  break;
+ case CONFIGFS_CTN_ATTR_FREEING:
+  break;
+ case CONFIGFS_CTN_ATTR_ADD_TASK:
+ case CONFIGFS_CTN_ATTR_RM_TASK:
+  {
+   struct task_struct *t;
+
+   read_lock(&tasklist_lock);
+   t = find_task_by_pid(tmp);
+   if (t) {
+    get_task_struct(t);
```

```
+    read_unlock(&tasklist_lock);
+    if (ctfs_attr->idx == CONFIGFS_CTN_ATTR_ADD_TASK)
+     tmp = container_add_task(t, NULL, &sc->ctn);
+    else
+     container_remove_task(t, &sc->ctn);
+    put_task_struct(t);
+   }
+   else
+    read_unlock(&tasklist_lock);
+   break;
+  }
+ default:
+  printk("Invalid set attr option %d\n", ctfs_attr->idx);
+ }
+
+ return count;
+}
+
+/*
+ * This is where the release operation of container will come when a
+ * container is getting removed from container directory.  We will just release
+ * the memory allocated for the container.
+ */
+static void simple_containerfs_release(struct config_item *item)
+{
+ struct simple_containerfs *sc = to_simple_containerfs(item);
+
+ sc->ctn.freeing = 1;
+ smp_mb();
+ free_container(&sc->ctn);
+ kfree(to_simple_containerfs(item));
+}
+
+static struct configfs_item_operations container_item_ops = {
+ .release = simple_containerfs_release,
+ .show_attribute = simple_containerfs_attr_show,
+ .store_attribute = simple_containerfs_attr_store,
+};
+
+static struct config_item_type simple_containerfs_type = {
+ .ct_attrs = simple_containerfs_attrs,
+ .ct_item_ops = &container_item_ops,
+ .ct_owner = THIS_MODULE,
+};
+
+struct containerfs {
+ struct config_group group;
+};
```

```
+
+static inline struct containerfs *to_containerfs(struct config_item *item)
+{
+ return container_of(to_config_group(item), struct containerfs, group);
+}
+
+/*
+ * Containers are initialized here.  mkdir command underneath /configfs/container
+ * will get here. name is the pointer to the container name (given as part of
+ * mkdir command.
+ */
+
+static struct config_item *containerfs_make_item(struct config_group *group, const char *name)
+{
+ struct simple_containerfs *sc;
+
+ sc = kzalloc(sizeof(struct simple_containerfs) + strlen(name) + 1,
+   GFP_KERNEL);
+ if (!sc)
+  return NULL;
+
+ mutex_init(&sc->ctn.mutex);
+ sc->ctn.name = (char *) sc + sizeof(struct simple_containerfs);
+ strcpy(sc->ctn.name, name);
+ sc->ctn.page_limit = DEFAULT_PAGE_LIMIT;
+ sc->ctn.task_limit = DEFAULT_TASK_LIMIT;
+
+ INIT_WORK(&sc->ctn.work, container_overlimit_handler, (void *)&sc->ctn);
+ init_waitqueue_head(&sc->ctn.mm_waitq);
+ INIT_LIST_HEAD(&sc->ctn.tasks);
+ INIT_LIST_HEAD(&sc->ctn.mappings);
+
+ if (setup_container(&sc->ctn) < 0) {
+  kfree(sc);
+  return NULL;
+ }
+
+ config_item_init_type_name(&sc->item, name, &simple_containerfs_type);
+
+ return &sc->item;
+}
+
+static struct configfs_attribute containerfs_attr_description = {
+ .ca_owner = THIS_MODULE,
+ .ca_name = "description",
+ .ca_mode = S_IRUGO ,
+};
+
```

```
+static struct configfs_attribute *containerfs_attrs[] = {
+ &containerfs_attr_description,
+ NULL,
+};
+
+
+/* This is the read operation on the top level /configfs/containers/description.
+ * A general description about containers ...
+ */
+
+static ssize_t containerfs_attr_show(struct config_item *item,
+  struct configfs_attribute *attr,
+  char *page)
+{
+ return sprintf(page, "Containers provide grouping of resources in a "
+   "platform.  It also provides limits and accounting of"
+   "resources\nas they are used by processes belonging to"
+   "those containers\n");
+}
+
+static void containerfs_release(struct config_item *item)
+{
+ kfree(to_containerfs(item));
+}
+
+static struct configfs_item_operations containerfs_item_ops = {
+ .release = containerfs_release,
+ .show_attribute = containerfs_attr_show,
+};
+
+
+static struct configfs_group_operations containerfs_group_ops = {
+ .make_item = containerfs_make_item,
+};
+
+static struct config_item_type containerfs_type = {
+ .ct_item_ops = &containerfs_item_ops,
+ .ct_group_ops = &containerfs_group_ops,
+ .ct_attrs = containerfs_attrs,
+ .ct_owner = THIS_MODULE,
+};
+
+struct containerfs_group {
+ struct configfs_subsystem cs_subsys;
+};
+
+static struct containerfs_group containerfs_group = {
+ .cs_subsys = {
```

```
+  .su_group = {
+   .cg_item = {
+    .ci_namebuf = "containers",
+    .ci_type = &containerfs_type,
+   },
+  },
+ },
+};
+
+static int __init configfs_container_init(void)
+{
+ int ret;
+
+  config_group_init(&containerfs_group.cs_subsys.su_group) ;
+ init_MUTEX(&containerfs_group.cs_subsys.su_sem);
+ ret =  configfs_register_subsystem(&containerfs_group.cs_subsys );
+
+ if (ret)
+  printk(KERN_ERR "Error %d while registering container subsystem\n", ret);
+ else {
+  container_wq = create_workqueue("Kcontainerd");
+  if (container_wq == NULL) {
+    configfs_unregister_subsystem(&containerfs_group.cs_subs ys);
+   ret = -ENOMEM;
+   printk(KERN_ERR "Unable to create Container controllers");
+  }
+ }
+ return ret;
+}
+
+/* Depends on configfs inititalization.
+ */
+late_initcall(configfs_container_init);
+
```