
Subject: [PATCH 10/10] Containers(V10): Support for automatic userspace release agents

Posted by [Paul Menage](#) on Tue, 29 May 2007 13:01:14 GMT

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

This patch adds the following files to the container filesystem:

notify_on_release - configures/reports whether the container subsystem should attempt to run a release script when this container becomes unused

release_agent - configures/reports the release agent to be used for this hierarchy (top level in each hierarchy only)

releasable - reports whether this container would have been auto-released if notify_on_release was true and a release agent was configured (mainly useful for debugging)

To avoid locking issues, invoking the userspace release agent is done via a workqueue task; containers that need to have their release agents invoked by the workqueue task are linked on to a list.

When the "cpuset" filesystem is mounted, it automatically sets the hierarchy's release agent to be /sbin/cpuset_release_agent for backward-compatibility with existing cpusets users.

Signed-off-by: Paul Menage <menage@google.com>

```
include/linux/container.h | 15 +
kernel/container.c        | 364 ++++++
kernel/cpuset.c           | 5
3 files changed, 348 insertions(+), 36 deletions(-)
```

Index: container-2.6.22-rc2-mm1/include/linux/container.h

```
=====
--- container-2.6.22-rc2-mm1.orig/include/linux/container.h
+++ container-2.6.22-rc2-mm1/include/linux/container.h
@@ -64,11 +64,7 @@ static inline void css_get(struct container_state *css)
 * css_put() should be called to release a reference taken by
 * css_get()
 */
-
-static inline void css_put(struct container_subsys_state *css)
-{
- atomic_dec(&css->refcnt);
-}
+void css_put(struct container_subsys_state *css);
```

```

struct container {
    unsigned long flags; /* "unsigned long" so bitops work */
@@ -99,6 +95,13 @@ struct container {
    * tasks in this container. Protected by css_group_lock
    */
    struct list_head css_groups;
+
+ /*
+ * Linked list running through all containers that can
+ * potentially be reaped by the release agent. Protected by
+ * container_mutex
+ */
+ struct list_head release_list;
};

/* A css_group is a structure holding pointers to a set of
@@ -271,6 +274,8 @@ struct task_struct *container_iter_next(
    struct container_iter *it);
void container_iter_end(struct container *cont, struct container_iter *it);

+void container_set_release_agent_path(struct container_subsys *ss,
+    const char *path);

#else /* !CONFIG_CONTAINERS */

```

Index: container-2.6.22-rc2-mm1/kernel/container.c

=====

--- container-2.6.22-rc2-mm1.orig/kernel/container.c

+++ container-2.6.22-rc2-mm1/kernel/container.c

@@ -62,6 +62,8 @@

```

#define CONTAINER_SUPER_MAGIC 0x27e0eb

```

```

+static DEFINE_MUTEX(container_mutex);
+

```

```

/* Generate an array of container subsystem pointers */

```

```

#define SUBSYS(_x) &_x ## _subsys,

```

@@ -89,6 +91,13 @@ struct containerfs_root {

```

/* A list running through the mounted hierarchies */

```

```

    struct list_head root_list;
+

```

```

+ /* The path to use for release notifications. No locking
+ * between setting and use - so if userspace updates this
+ * while subcontainers exist, you could miss a
+ * notification. We ensure that it's always a valid
+ * NUL-terminated string */

```

```

+ char release_agent_path[PATH_MAX];
};

@@ -115,7 +124,13 @@ static int need_forkexit_callback = 0;

/* bits in struct container flags field */
typedef enum {
+ /* Container is dead */
  CONT_REMOVED,
+ /* Container has previously had a child container or a task,
+  * but no longer (only if CONT_NOTIFY_ON_RELEASE is set) */
+ CONT_RELEASABLE,
+ /* Container requires release notifications to userspace */
+ CONT_NOTIFY_ON_RELEASE,
} container_flagbits_t;

/* convenient tests for these bits */
@@ -124,6 +139,19 @@ inline int container_is_removed(const st
    return test_bit(CONT_REMOVED, &cont->flags);
}

+inline int container_is_releasable(const struct container *cont)
+{
+ const int bits =
+ (1 << CONT_RELEASABLE) |
+ (1 << CONT_NOTIFY_ON_RELEASE);
+ return (cont->flags & bits) == bits;
+}
+
+inline int notify_on_release(const struct container *cont)
+{
+ return test_bit(CONT_NOTIFY_ON_RELEASE, &cont->flags);
+}
+
/* for_each_subsys() allows you to iterate on each subsystem attached to
 * an active hierarchy */
#define for_each_subsys(_root, _ss) \
@@ -133,6 +161,12 @@ list_for_each_entry(_ss, &_root->subsys_
#define for_each_root(_root) \
list_for_each_entry(_root, &roots, root_list)

+/* the list of containers eligible for automatic release */
+static LIST_HEAD(release_list);
+static void container_release_agent(struct work_struct *work);
+static DECLARE_WORK(release_agent_work, container_release_agent);
+static void check_for_release(struct container *cont);
+

```

```

/* Link structure for associating css_group objects with containers */
struct cg_container_link {
/*
@@ -181,11 +215,8 @@ static int css_group_count;
/*
 * unlink a css_group from the list and free it
 */
-static void release_css_group(struct kref *k)
+static void unlink_css_group(struct css_group *cg)
{
- struct css_group *cg =
- container_of(k, struct css_group, ref);
- int i;
- write_lock(&css_group_lock);
- list_del(&cg->list);
- css_group_count--;
@@ -198,8 +229,47 @@ static void release_css_group(struct kre
- kfree(link);
- }
- write_unlock(&css_group_lock);
+}
+
+static void release_css_group(struct kref *k)
+{
+ int i;
+ struct css_group *cg = container_of(k, struct css_group, ref);
+ BUG_ON(!mutex_is_locked(&container_mutex));
+
+ unlink_css_group(cg);
+ for (i = 0; i < CONTAINER_SUBSYS_COUNT; i++) {
+ atomic_dec(&cg->subsys[i]->container->count);
+ struct container *cont = cg->subsys[i]->container;
+ if (atomic_dec_and_test(&cont->count) &&
+ container_is_releasable(cont)) {
+ check_for_release(cont);
+ }
+ }
+ kfree(cg);
+}
+
+/*
+ * In the task exit path we want to avoid taking container_mutex
+ * unless absolutely necessary, so the release process is slightly
+ * different.
+ */
+static void release_css_group_taskexit(struct kref *k)
+{
+ int i;

```

```

+ struct css_group *cg = container_of(k, struct css_group, ref);
+ unlink_css_group(cg);
+ for (i = 0; i < CONTAINER_SUBSYS_COUNT; i++) {
+ struct container *cont = cg->subsys[i]->container;
+ if (notify_on_release(cont)) {
+ mutex_lock(&container_mutex);
+ set_bit(CONT_RELEASABLE, &cont->flags);
+ if (atomic_dec_and_test(&cont->count)) {
+ check_for_release(cont);
+ }
+ mutex_unlock(&container_mutex);
+ } else {
+ atomic_dec(&cont->count);
+ }
+ }
+ kfree(cg);
+ }
@@ -217,6 +287,11 @@ static inline void put_css_group(struct
+ kref_put(&cg->ref, release_css_group);
+ }

+static inline void put_css_group_taskexit(struct css_group *cg)
+{
+ kref_put(&cg->ref, release_css_group_taskexit);
+}
+
+/*
+ * find_existing_css_group() is a helper for
+ * find_css_group(), and checks to see whether an existing
@@ -446,8 +521,6 @@ static struct css_group *find_css_group(
+ * update of a tasks container pointer by attach_task()
+ */

-static DEFINE_MUTEX(container_mutex);
-
-/**
+ * container_lock - lock out any changes to container structures
+ */
@@ -795,6 +868,7 @@ static int container_fill_super(struct s
+ root->d_fsdata = &hroot->top_container;
+ hroot->top_container.dentry = root;

+ strcpy(hroot->release_agent_path, "");
+ sb->s_fs_info = hroot;
+ hroot->sb = sb;

@@ -811,6 +885,7 @@ static void init_container_root(struct c
+ INIT_LIST_HEAD(&cont->sibling);

```

```

    INIT_LIST_HEAD(&cont->children);
    INIT_LIST_HEAD(&cont->css_groups);
+ INIT_LIST_HEAD(&cont->release_list);
    list_add(&root->root_list, &roots);
    root_count++;
}
@@ -1057,7 +1132,7 @@ static int attach_task(struct container
    ss->attach(ss, cont, oldcont, tsk);
}
}
-
+ set_bit(CONT_RELEASABLE, &oldcont->flags);
    synchronize_rcu();
    put_css_group(cg);
    return 0;
@@ -1109,6 +1184,9 @@ typedef enum {
    FILE_ROOT,
    FILE_DIR,
    FILE_TASKLIST,
+ FILE_NOTIFY_ON_RELEASE,
+ FILE_RELEASABLE,
+ FILE_RELEASE_AGENT,
} container_filetype_t;

static ssize_t container_common_file_write(struct container *cont,
@@ -1145,6 +1223,28 @@ static ssize_t container_common_file_wri
    case FILE_TASKLIST:
        retval = attach_task_by_pid(cont, buffer);
        break;
+ case FILE_NOTIFY_ON_RELEASE:
+     clear_bit(CONT_RELEASABLE, &cont->flags);
+     if (simple_strtoul(buffer, NULL, 10) != 0)
+         set_bit(CONT_NOTIFY_ON_RELEASE, &cont->flags);
+     else
+         clear_bit(CONT_NOTIFY_ON_RELEASE, &cont->flags);
+     break;
+ case FILE_RELEASE_AGENT:
+ {
+     struct containerfs_root *root = cont->root;
+     if (nbytes < sizeof(root->release_agent_path)) {
+         /* We never write anything other than '\0'
+          * into the last char of release_agent_path,
+          * so it always remains a NUL-terminated
+          * string */
+         strncpy(root->release_agent_path, buffer, nbytes);
+         root->release_agent_path[nbytes] = 0;
+     } else {
+         retval = -ENOSPC;

```

```

+ }
+ break;
+ }
  default:
    retval = -EINVAL;
    goto out2;
@@ -1183,6 +1283,49 @@ static ssize_t container_read_uint(struc
  return simple_read_from_buffer(buf, nbytes, ppos, tmp, len);
}

+static ssize_t container_common_file_read(struct container *cont,
+    struct cftype *cft,
+    struct file *file,
+    char __user *buf,
+    size_t nbytes, loff_t *ppos)
+{
+ container_filetype_t type = cft->private;
+ char *page;
+ ssize_t retval = 0;
+ char *s;
+
+ if (!(page = (char *)__get_free_page(GFP_KERNEL)))
+ return -ENOMEM;
+
+ s = page;
+
+ switch (type) {
+ case FILE_RELEASE_AGENT:
+ {
+ struct containerfs_root *root;
+ size_t n;
+ mutex_lock(&container_mutex);
+ root = cont->root;
+ n = strlen(root->release_agent_path,
+     sizeof(root->release_agent_path));
+ n = min(n, (size_t) PAGE_SIZE);
+ strncpy(s, root->release_agent_path, n);
+ mutex_unlock(&container_mutex);
+ s += n;
+ break;
+ }
+ default:
+ retval = -EINVAL;
+ goto out;
+ }
+ *s++ = '\n';
+
+ retval = simple_read_from_buffer(buf, nbytes, ppos, page, s - page);

```

```

+out:
+ free_page((unsigned long)page);
+ return retval;
+}
+
+static ssize_t container_file_read(struct file *file, char __user *buf,
+    size_t nbytes, loff_t *ppos)
+{
@@ -1560,17 +1703,51 @@ static int container_tasks_release(struc
+    return 0;
+}

+static u64 container_read_notify_on_release(struct container *cont,
+    struct cftype *cft)
+{
+    return notify_on_release(cont);
+}
+
+static u64 container_read_releasable(struct container *cont,
+    struct cftype *cft)
+{
+    return test_bit(CONT_RELEASABLE, &cont->flags);
+}
+
+/*
+ * for the common functions, 'private' gives the type of file
+ */

-static struct cftype cft_tasks = {
-    .name = "tasks",
-    .open = container_tasks_open,
-    .read = container_tasks_read,
+static struct cftype files[] = {
+    {
+        .name = "tasks",
+        .open = container_tasks_open,
+        .read = container_tasks_read,
+        .write = container_common_file_write,
+        .release = container_tasks_release,
+        .private = FILE_TASKLIST,
+    },
+    {
+        .name = "notify_on_release",
+        .read_uint = container_read_notify_on_release,
+        .write = container_common_file_write,
+        .private = FILE_NOTIFY_ON_RELEASE,
+    },

```

```

+
+ {
+ .name = "releasable",
+ .read_uint = container_read_releasable,
+ .private = FILE_RELEASABLE,
+ }
+};
+
+static struct cftype cft_release_agent = {
+ .name = "release_agent",
+ .read = container_common_file_read,
+ .write = container_common_file_write,
+ .release = container_tasks_release,
+ .private = FILE_TASKLIST,
+ .private = FILE_RELEASE_AGENT,
+ };

static int container_populate_dir(struct container *cont)
@@ -1581,9 +1758,14 @@ static int container_populate_dir(struct
/* First clear out any existing files */
container_clear_directory(cont->dentry);

- if ((err = container_add_file(cont, &cft_tasks)) < 0)
+ if ((err = container_add_files(cont, files, ARRAY_SIZE(files)) < 0))
return err;

+ if (cont == cont->top_container) {
+ if ((err = container_add_file(cont, &cft_release_agent)) < 0)
+ return err;
+ }
+
for_each_subsys(cont->root, ss) {
if (ss->populate && (err = ss->populate(ss, cont)) < 0)
return err;
@@ -1635,6 +1817,7 @@ static long container_create(struct cont
INIT_LIST_HEAD(&cont->sibling);
INIT_LIST_HEAD(&cont->children);
INIT_LIST_HEAD(&cont->css_groups);
+ INIT_LIST_HEAD(&cont->release_list);

cont->parent = parent;
cont->root = parent->root;
@@ -1693,6 +1876,24 @@ static int container_mkdir(struct inode
return container_create(c_parent, dentry, mode | S_IFDIR);
}

+static inline int container_has_css_refs(struct container *cont)
+{

```

```

+ /* Check the reference count on each subsystem. Since we
+ * already established that there are no tasks in the
+ * container, if the css refcount is also 0, then there should
+ * be no outstanding references, so the subsystem is safe to
+ * destroy */
+ struct container_subsys *ss;
+ for_each_subsys(cont->root, ss) {
+ struct container_subsys_state *css;
+ css = cont->subsys[ss->subsys_id];
+ if (atomic_read(&css->refcnt)) {
+ return 1;
+ }
+ }
+ return 0;
+}
+
static int container_rmdir(struct inode *unused_dir, struct dentry *dentry)
{
    struct container *cont = dentry->d_fsdata;
@@ -1701,7 +1902,6 @@ static int container_rmdir(struct inode
    struct container_subsys *ss;
    struct super_block *sb;
    struct containerfs_root *root;
- int css_busy = 0;

    /* the vfs holds both inode->i_mutex already */

@@ -1719,20 +1919,7 @@ static int container_rmdir(struct inode
    root = cont->root;
    sb = root->sb;

- /* Check the reference count on each subsystem. Since we
- * already established that there are no tasks in the
- * container, if the css refcount is also 0, then there should
- * be no outstanding references, so the subsystem is safe to
- * destroy */
- for_each_subsys(root, ss) {
- struct container_subsys_state *css;
- css = cont->subsys[ss->subsys_id];
- if (atomic_read(&css->refcnt)) {
- css_busy = 1;
- break;
- }
- }
- if (css_busy) {
+ if (container_has_css_refs(cont)) {
    mutex_unlock(&container_mutex);
    return -EBUSY;

```

```

}
@@ -1754,6 +1941,11 @@ static int container_rmdir(struct inode
    dput(d);
    root->number_of_containers--;

+ if (!list_empty(&cont->release_list))
+ list_del(&cont->release_list);
+ set_bit(CONT_RELEASABLE, &parent->flags);
+ check_for_release(parent);
+
    mutex_unlock(&container_mutex);
    /* Drop the active superblock reference that we took when we
     * created the container */
@@ -2107,7 +2299,7 @@ void container_exit(struct task_struct *
    tsk->containers = &init_css_group;
    task_unlock(tsk);
    if (cg)
- put_css_group(cg);
+ put_css_group_taskexit(cg);
}

static atomic_t namecnt;
@@ -2215,7 +2407,10 @@ int container_clone(struct task_struct *

    out_release:
    mutex_unlock(&inode->i_mutex);
+
+ mutex_lock(&container_mutex);
    put_css_group(cg);
+ mutex_unlock(&container_mutex);
    deactivate_super(parent->root->sb);
    return ret;
}
@@ -2236,3 +2431,110 @@ int container_is_descendant(const struct
    ret = (cont == target);
    return ret;
}
+
+static void check_for_release(struct container *cont)
+{
+ BUG_ON(!mutex_is_locked(&container_mutex));
+ if (container_is_releasable(cont) && !atomic_read(&cont->count)
+ && list_empty(&cont->children) && !container_has_css_refs(cont)) {
+ /* Container is currently removeable. If it's not
+  * already queued for a userspace notification, queue
+  * it now */
+ if (list_empty(&cont->release_list)) {
+ list_add(&cont->release_list, &release_list);

```

```

+ schedule_work(&release_agent_work);
+ }
+ }
+}
+
+void css_put(struct container_subsys_state *css)
+{
+ struct container *cont = css->container;
+ if (notify_on_release(cont)) {
+ mutex_lock(&container_mutex);
+ set_bit(CONT_RELEASABLE, &cont->flags);
+ if (atomic_dec_and_test(&css->refcnt)) {
+ check_for_release(cont);
+ }
+ mutex_unlock(&container_mutex);
+ } else {
+ atomic_dec(&css->refcnt);
+ }
+}
+
+void container_set_release_agent_path(struct container_subsys *ss,
+      const char *path)
+{
+ mutex_lock(&container_mutex);
+ strcpy(ss->root->release_agent_path, path);
+ mutex_unlock(&container_mutex);
+}
+
+/*
+ * Notify userspace when a container is released, by running the
+ * configured release agent with the name of the container (path
+ * relative to the root of container file system) as the argument.
+ *
+ * Most likely, this user command will try to rmdir this container.
+ *
+ * This races with the possibility that some other task will be
+ * attached to this container before it is removed, or that some other
+ * user task will 'mkdir' a child container of this container. That's ok.
+ * The presumed 'rmdir' will fail quietly if this container is no longer
+ * unused, and this container will be reprieved from its death sentence,
+ * to continue to serve a useful existence. Next time it's released,
+ * we will get notified again, if it still has 'notify_on_release' set.
+ *
+ * The final arg to call_usermodehelper() is UMH_WAIT_EXEC, which
+ * means only wait until the task is successfully execve()'d. The
+ * separate release agent task is forked by call_usermodehelper(),
+ * then control in this thread returns here, without waiting for the
+ * release agent task. We don't bother to wait because the caller of

```

```

+ * this routine has no use for the exit status of the release agent
+ * task, so no sense holding our caller up for that.
+ *
+ */
+
+static void container_release_agent(struct work_struct *work)
+{
+ BUG_ON(work != &release_agent_work);
+ mutex_lock(&container_mutex);
+ while (!list_empty(&release_list)) {
+ char *argv[3], *envp[3];
+ int i;
+ char *pathbuf;
+ struct container *cont = list_entry(release_list.next,
+ struct container,
+ release_list);
+ list_del_init(&cont->release_list);
+
+ pathbuf = kmalloc(PAGE_SIZE, GFP_KERNEL);
+ if (!pathbuf)
+ continue;
+
+ if (container_path(cont, pathbuf, PAGE_SIZE) < 0) {
+ kfree(pathbuf);
+ continue;
+ }
+
+ i = 0;
+ argv[i++] = cont->root->release_agent_path;
+ argv[i++] = (char *)pathbuf;
+ argv[i] = NULL;
+
+ i = 0;
+ /* minimal command environment */
+ envp[i++] = "HOME=";
+ envp[i++] = "PATH=/sbin:/bin:/usr/sbin:/usr/bin";
+ envp[i] = NULL;
+
+ /* Drop the lock while we invoke the usermode helper,
+ * since the exec could involve hitting disk and hence
+ * be a slow process */
+ mutex_unlock(&container_mutex);
+ call_usermodehelper(argv[0], argv, envp, UMH_WAIT_EXEC);
+ kfree(pathbuf);
+ mutex_lock(&container_mutex);
+ }
+ mutex_unlock(&container_mutex);
+}

```

Index: container-2.6.22-rc2-mm1/kernel/cpuset.c

=====

--- container-2.6.22-rc2-mm1.orig/kernel/cpuset.c

+++ container-2.6.22-rc2-mm1/kernel/cpuset.c

```
@@ -275,6 +275,11 @@ static int cpuset_get_sb(struct file_sys
    unused_dev_name,
    "cpuset", mnt);
    put_filesystem(container_fs);
+ if (!ret) {
+   container_set_release_agent_path(
+     &cpuset_subsys,
+     "/sbin/cpuset_release_agent");
+ }
+ }
    return ret;
}
```

--
