
Subject: Re: [PATCH 07/10] Task Containers(V11): Automatic userspace notification of idle containers

Posted by [serue](#) on Mon, 23 Jul 2007 17:41:02 GMT

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

Quoting menage@google.com (menage@google.com):

> 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.

Hi Paul,

my tree may be a bit crufty, but I had to `#include <linux/kmod.h>` in order for this to compile on s390.

thanks,
-serge

> Signed-off-by: Paul Menage <menage@google.com>
> ---
>
> include/linux/container.h | 11 -
> kernel/container.c | 425 ++++++-----
> 2 files changed, 393 insertions(+), 43 deletions(-)
>
> Index: container-2.6.22-rc6-mm1/include/linux/container.h
> =====
> --- container-2.6.22-rc6-mm1.orig/include/linux/container.h
> +++ container-2.6.22-rc6-mm1/include/linux/container.h
> @@ -77,10 +77,11 @@ static inline void css_get(struct contain
> * css_get()
> */
>
> +extern void __css_put(struct container_subsys_state *css);
> static inline void css_put(struct container_subsys_state *css)

```

> {
> if (!test_bit(CSS_ROOT, &css->flags))
> - atomic_dec(&css->refcnt);
> + __css_put(css);
> }
>
> struct container {
> @@ -112,6 +113,13 @@ 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
> + * release_list_lock
> + */
> + struct list_head release_list;
> };
>
> /* A css_group is a structure holding pointers to a set of
> @@ -285,7 +293,6 @@ struct task_struct *container_iter_next(
> struct container_iter *it);
> void container_iter_end(struct container *cont, struct container_iter *it);
>
> -
> #else /* !CONFIG_CONTAINERS */
>
> static inline int container_init_early(void) { return 0; }
> Index: container-2.6.22-rc6-mm1/kernel/container.c
> =====
> --- container-2.6.22-rc6-mm1.orig/kernel/container.c
> +++ container-2.6.22-rc6-mm1/kernel/container.c
> @@ -44,6 +44,8 @@
> #include <linux/sort.h>
> #include <asm/atomic.h>
>
> +static DEFINE_MUTEX(container_mutex);
> +
> /* Generate an array of container subsystem pointers */
> #define SUBSYS(_x) &_x ## _subsys,
>
> @@ -82,6 +84,13 @@ struct containerfs_root {
>
> /* Hierarchy-specific flags */
> unsigned long flags;
> +
> + /* 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];
> };
>
>
> @@ -109,7 +118,13 @@ static int need_forkexit_callback;
>
> /* bits in struct container flags field */
> 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,
> };
>
> /* convenient tests for these bits */
> @@ -123,6 +138,19 @@ enum {
> ROOT_NOPREFIX, /* mounted subsystems have no named prefix */
> };
>
> +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
> @@ -134,6 +162,14 @@ 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. Protected by
> + * release_list_lock */

```

```

> +static LIST_HEAD(release_list);
> +static DEFINE_SPINLOCK(release_list_lock);
> +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 {
> /*
> @@ -188,11 +224,8 @@ static int use_task_css_group_links;
> /*
> * 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--;
> @@ -205,11 +238,39 @@ static void release_css_group(struct kre
> kfree(link);
> }
> write_unlock(&css_group_lock);
> - for (i = 0; i < CONTAINER_SUBSYS_COUNT; i++)
> - atomic_dec(&cg->subsys[i]->container->count);
> +}
> +
> +static void __release_css_group(struct kref *k, int taskexit)
> +{
> + int i;
> + struct css_group *cg = container_of(k, struct css_group, ref);
> +
> + unlink_css_group(cg);
> +
> + rcu_read_lock();
> + for (i = 0; i < CONTAINER_SUBSYS_COUNT; i++) {
> + struct container *cont = cg->subsys[i]->container;
> + if (atomic_dec_and_test(&cont->count) &&
> + notify_on_release(cont)) {
> + if (taskexit)
> + set_bit(CONT_RELEASABLE, &cont->flags);
> + check_for_release(cont);
> + }
> + }
> + rcu_read_unlock();

```

```

> kfree(cg);
> }
>
> +static void release_css_group(struct kref *k)
> +{
> + __release_css_group(k, 0);
> +}
> +
> +static void release_css_group_taskexit(struct kref *k)
> +{
> + __release_css_group(k, 1);
> +}
> +
> /*
> * refcounted get/put for css_group objects
> */
> @@ -223,6 +284,11 @@ 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
> @@ -464,8 +530,6 @@ 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
> *
> @@ -524,6 +588,13 @@ static void container_diput(struct dentr
> if (S_ISDIR(inode->i_mode)) {
> struct container *cont = dentry->d_fsdata;
> BUG_ON(!(container_is_removed(cont)));
> + /* It's possible for external users to be holding css
> + * reference counts on a container; css_put() needs to
> + * be able to access the container after decrementing
> + * the reference count in order to know if it needs to
> + * queue the container to be handled by the release
> + * agent */
> + synchronize_rcu();

```

```

> kfree(cont);
> }
> iput(inode);
> @@ -668,6 +739,8 @@ static int container_show_options(struct
> seq_printf(seq, "%s", ss->name);
> if (test_bit(ROOT_NOPREFIX, &root->flags))
> seq_puts(seq, "noprefix");
> + if (strlen(root->release_agent_path))
> + seq_printf(seq, "release_agent=%s", root->release_agent_path);
> mutex_unlock(&container_mutex);
> return 0;
> }
> @@ -675,6 +748,7 @@ static int container_show_options(struct
> struct container_sb_opts {
> unsigned long subsys_bits;
> unsigned long flags;
> + char *release_agent;
> };
>
> /* Convert a hierarchy specifier into a bitmask of subsystems and
> @@ -686,6 +760,7 @@ static int parse_containerfs_options(char
>
> opts->subsys_bits = 0;
> opts->flags = 0;
> + opts->release_agent = NULL;
>
> while ((token = strsep(&o, ",")) != NULL) {
> if (!*token)
> @@ -694,6 +769,15 @@ static int parse_containerfs_options(char
> opts->subsys_bits = (1 << CONTAINER_SUBSYS_COUNT) - 1;
> } else if (!strcmp(token, "noprefix")) {
> set_bit(ROOT_NOPREFIX, &opts->flags);
> + } else if (!strncmp(token, "release_agent=", 14)) {
> + /* Specifying two release agents is forbidden */
> + if (opts->release_agent)
> + return -EINVAL;
> + opts->release_agent = kzalloc(PATH_MAX, GFP_KERNEL);
> + if (!opts->release_agent)
> + return -ENOMEM;
> + strncpy(opts->release_agent, token + 14, PATH_MAX - 1);
> + opts->release_agent[PATH_MAX - 1] = 0;
> } else {
> struct container_subsys *ss;
> int i;
> @@ -743,7 +827,11 @@ static int container_remount(struct super
> if (!ret)
> container_populate_dir(cont);
>

```

```

> + if (opts.release_agent)
> + strcpy(root->release_agent_path, opts.release_agent);
> out_unlock:
> + if (opts.release_agent)
> + kfree(opts.release_agent);
> mutex_unlock(&container_mutex);
> mutex_unlock(&cont->dentry->d_inode->i_mutex);
> return ret;
> @@ -767,6 +855,7 @@ 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);
> }
>
> static int container_test_super(struct super_block *sb, void *data)
> @@ -841,8 +930,11 @@ static int container_get_sb(struct file_
>
> /* First find the desired set of subsystems */
> ret = parse_containerfs_options(data, &opts);
> - if (ret)
> + if (ret) {
> + if (opts.release_agent)
> + kfree(opts.release_agent);
> return ret;
> + }
>
> root = kzalloc(sizeof(*root), GFP_KERNEL);
> if (!root)
> @@ -851,6 +943,10 @@ static int container_get_sb(struct file_
> init_container_root(root);
> root->subsys_bits = opts.subsys_bits;
> root->flags = opts.flags;
> + if (opts.release_agent) {
> + strcpy(root->release_agent_path, opts.release_agent);
> + kfree(opts.release_agent);
> + }
>
> sb = sget(fs_type, container_test_super, container_set_super, root);
>
> @@ -1125,7 +1221,7 @@ 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;
> @@ -1175,6 +1271,9 @@ enum container_filetype {
> FILE_ROOT,
> FILE_DIR,
> FILE_TASKLIST,
> + FILE_NOTIFY_ON_RELEASE,
> + FILE_RELEASABLE,
> + FILE_RELEASE_AGENT,
> };
>
> static ssize_t container_common_file_write(struct container *cont,
> @@ -1212,6 +1311,32 @@ 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;
> +     /* Strip trailing newline */
> +     if (nbytes && (buffer[nbytes-1] == '\n')) {
> +         buffer[nbytes-1] = 0;
> +     }
> +     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;
> @@ -1252,6 +1377,49 @@ 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)
> +{
> + enum container_filetype 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)
> {
> @@ -1667,16 +1835,49 @@ 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)
> @@ -1687,10 +1888,15 @@ static int container_populate_dir(struct
> /* First clear out any existing files */
> container_clear_directory(cont->dentry);
>
> - err = container_add_file(cont, NULL, &cft_tasks);
> + err = container_add_files(cont, NULL, files, ARRAY_SIZE(files));
> if (err < 0)
> return err;
>
> + if (cont == cont->top_container) {
> + if ((err = container_add_file(cont, NULL, &cft_release_agent)) < 0)
> + return err;
> + }
> +
> for_each_subsys(cont->root, ss) {
> if (ss->populate && (err = ss->populate(ss, cont)) < 0)
> return err;
> @@ -1747,6 +1953,7 @@ 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;
> @@ -1808,6 +2015,38 @@ 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. We scan across all subsystems rather than using
> + * the per-hierarchy linked list of mounted subsystems since
> + * we can be called via check_for_release() with no
> + * synchronization other than RCU, and the subsystem linked
> + * list isn't RCU-safe */
> + int i;
> + for (i = 0; i < CONTAINER_SUBSYS_COUNT; i++) {

```

```

> + struct container_subsys *ss = subsys[i];
> + struct container_subsys_state *css;
> + /* Skip subsystems not in this hierarchy */
> + if (ss->root != cont->root)
> +     continue;
> + css = cont->subsys[ss->subsys_id];
> + /* When called from check_for_release() it's possible
> +  * that by this point the container has been removed
> +  * and the css deleted. But a false-positive doesn't
> +  * matter, since it can only happen if the container
> +  * has been deleted and hence no longer needs the
> +  * release agent to be called anyway. */
> + if (css && 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;
> @@ -1816,7 +2055,6 @@ 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 */
>
> @@ -1834,20 +2072,7 @@ 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;
> }
> @@ -1857,7 +2082,11 @@ static int container_rmdir(struct inode
>     ss->destroy(ss, cont);
> }
>
> + spin_lock(&release_list_lock);
>     set_bit(CONT_REMOVED, &cont->flags);
> + if (!list_empty(&cont->release_list))
> +     list_del(&cont->release_list);
> + spin_unlock(&release_list_lock);
>     /* delete my sibling from parent->children */
>     list_del(&cont->sibling);
>     spin_lock(&cont->dentry->d_lock);
> @@ -1869,6 +2098,9 @@ static int container_rmdir(struct inode
>     dput(d);
>     root->number_of_containers--;
>
> + 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 */
> @@ -1906,15 +2138,15 @@ static void container_init_subsys(struct
>     /* If this subsystem requested that it be notified with fork
>      * events, we should send it one now for every process in the
>      * system */
> - if (ss->fork) {
> -     struct task_struct *g, *p;
> + if (ss->fork) {
> +     struct task_struct *g, *p;
>
> -     read_lock(&tasklist_lock);
> -     do_each_thread(g, p) {
> -         ss->fork(ss, p);
> -     } while_each_thread(g, p);
> -     read_unlock(&tasklist_lock);
> - }
> + read_lock(&tasklist_lock);
> + do_each_thread(g, p) {
> +     ss->fork(ss, p);
> + } while_each_thread(g, p);
> + read_unlock(&tasklist_lock);
> + }
>

```

```

> need_forkexit_callback |= ss->fork || ss->exit;
>
> @@ -2241,7 +2473,7 @@ 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);
> }
>
> /**
> @@ -2352,7 +2584,10 @@ 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;
> }
> @@ -2382,3 +2617,111 @@ int container_is_descendant(const struct
> ret = (cont == target);
> return ret;
> }
> +
> +static void check_for_release(struct container *cont)
> +{
> + /* All of these checks rely on RCU to keep the container
> + * structure alive */
> + 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 */
> + int need_schedule_work = 0;
> + spin_lock(&release_list_lock);
> + if (!container_is_removed(cont) &&
> + list_empty(&cont->release_list)) {
> + list_add(&cont->release_list, &release_list);
> + need_schedule_work = 1;
> + }
> + spin_unlock(&release_list_lock);
> + if (need_schedule_work)
> + schedule_work(&release_agent_work);
> + }
> +}

```

```

> +
> +void __css_put(struct container_subsys_state *css)
> +{
> + struct container *cont = css->container;
> + rcu_read_lock();
> + if (atomic_dec_and_test(&css->refcnt) && notify_on_release(cont)) {
> + set_bit(CONT_RELEASABLE, &cont->flags);
> + check_for_release(cont);
> + }
> + rcu_read_unlock();
> +}
> +
> +/*
> + * 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);
> + spin_lock(&release_list_lock);
> + 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);
> + spin_unlock(&release_list_lock);
> + pathbuf = kmalloc(PAGE_SIZE, GFP_KERNEL);
> + if (!pathbuf) {
> +     spin_lock(&release_list_lock);
> +     continue;
> + }
> +
> + if (container_path(cont, pathbuf, PAGE_SIZE) < 0) {
> +     kfree(pathbuf);
> +     spin_lock(&release_list_lock);
> +     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);
> + spin_lock(&release_list_lock);
> + }
> + spin_unlock(&release_list_lock);
> + mutex_unlock(&container_mutex);
> +}
>
> --

```
