Subject: [RFC/PATCH 0/8]: CGroup Files: Clean up locking and boilerplate
Posted by Paul Menage on Tue, 13 May 2008 06:37:07 GMT
View Forum Message <> Reply to Message

Most of the remaining cgroup control files that implement raw file handlers (those where the userspace pointer/length/ppos are passed through unchanged by cgroups) appear to do so in order to have some control over the kind of locking that their handler uses. In particular, some handlers need to call cgroup_lock() after copying data from userspace and others don't. They also often dynamically allocate memory when it's not strictly needed.

This patchset provides three main features:

1) A new "lockmode" field that indicates what kind of stability/locking guarantees the handler needs from cgroups while it is running. This can be used to reduce the dependency on cgroup_lock(), and prevent its use as a BKL from becoming a contention point as the number of cgroups subsystems grow.

An alternative to this would be to not have cgroups do the locking internally, but to export similar functionality via functions such as

cgroup_lock_rmdir(struct cgroup*)
cgroup_lock_hierarchy(struct cgroup*)
cgroup_lock_attach(struct cgroup*)

that would perform any necessary locking, and recheck !cgroup_removed(). This results in clearer code from the locking point of view, as you can see within the function exactly what locking it relies upon. The downside is that it makes the code more complex in that you can't simply return from the middle of the function, but instead need to save the return code and goto the point in the function that releases the lock. E.g. the difference would be:

```
int some_handler(struct cgroup *cgrp, struct cftype *cft)
{
 if (!do_locked_operation1(cgrp))
  return -EINVAL;
 return do_locked_operation2(cgrp);
}
```
(and setting CFT_LOCK_RMDIR in the cftype descriptor)

versus

```
int some_handler(struct cgroup *cgrp, struct cftype *cft)
{
 int retval = cgroup_lock_rmdir(cgrp);
```

```
 if (retval)
  return retval;
 if (!do_locked_operation1(cgrp)) {
  retval = -EINVAL;
  goto out_unlock;
 }
 retval = do_locked_operation2(cgrp);
out_unlock:
 cgroup_unlock_rmdir(cgrp);
 return retval;
}
```

The latter style is common in the kernel, but my feeling is that the former style is easier to code and to comprehend. I'm interested in what others think.

2) A new "write_string" method which copies the user's data to kernel space and ensures it's nul-terminated, performs any necessary locking/checks, and invokes the handler with the kernelspace buffer

3) Conversion of several raw read/write handlers in cgroup, cpuset, devcgroup and res_counter to use typed handlers and the new locking specifications.

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

--

Subject: [RFC/PATCH 1/8]: CGroup Files: Add locking mode to cgroups control files
Posted by Paul Menage on Tue, 13 May 2008 06:37:08 GMT
View Forum Message <> Reply to Message

Different cgroup files have different stability requirements of the cgroups framework while the handler is running; currently most subsystems that don't have their own internal synchronization just call cgroup_lock()/cgroup_unlock(), which takes the global cgroup_mutex.

This patch introduces a range of locking modes that can be requested by a control file; currently these are all implemented internally by taking cgroup_mutex, but expressing the intention will make it simpler to move to a finer-grained locking scheme in the future.

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

```
---
 include/linux/cgroup.h |   76 ++++++++++++++++++++
 kernel/cgroup.c        |  192 +++++++++++++++++++++++++++++++++++++++---------
 2 files changed, 212 insertions(+), 56 deletions(-)

Index: cgroup-2.6.25-mm1/include/linux/cgroup.h
===================================================================
--- cgroup-2.6.25-mm1.orig/include/linux/cgroup.h
+++ cgroup-2.6.25-mm1/include/linux/cgroup.h
@@ -200,11 +200,87 @@ struct cgroup_map_cb {
  */

 #define MAX_CFTYPE_NAME 64
+
+/* locking modes for control files.
+ *
+ * These determine what level of guarantee the file handler wishes
+ * cgroups to provide about the stability of control group entities
+ * for the duration of the handler callback.
+ *
+ * The minimum guarantee is that the subsystem state for this
+ * subsystem will not be freed (via a call to the subsystem's
+ * destroy() callback) until after the control file handler
+ * returns. This guarantee is provided by the fact that the open
+ * dentry for the control file keeps its parent (cgroup) dentry alive,
+ * which in turn keeps the cgroup object from being actually freed
+ * (although it can be moved into the removed state in the
+ * meantime). This is suitable for subsystems that completely control
+ * their own synchronization.
+ *
+ * Other possible guarantees are given below.
+ *
+ * XXX_READ bits are used for a read operation on the control file,
+ * XXX_WRITE bits are used for a write operation on the control file
+ */
+
+/*
+ * CFT_LOCK_ATTACH_(READ|WRITE): This operation will not run
+ * concurrently with a task movement into or out of this cgroup.
+ */
+#define CFT_LOCK_ATTACH_READ 1
+#define CFT_LOCK_ATTACH_WRITE 2
+#define CFT_LOCK_ATTACH (CFT_LOCK_ATTACH_READ | CFT_LOCK_ATTACH_WRITE)
+
+/*
+ * CFT_LOCK_RMDIR_(READ|WRITE): This operation will not run
+ * concurrently with the removal of the affected cgroup.
```

```
+ */
+#define CFT_LOCK_RMDIR_READ 4
+#define CFT_LOCK_RMDIR_WRITE 8
+#define CFT_LOCK_RMDIR (CFT_LOCK_RMDIR_READ | CFT_LOCK_RMDIR_WRITE)
+
+/*
+ * CFT_LOCK_HIERARCHY_(READ|WRITE): This operation will not run
+ * concurrently with a cgroup creation or removal in this hierarchy,
+ * or a bind/move/unbind for this subsystem.
+ */
+#define CFT_LOCK_HIERARCHY_READ 16
+#define CFT_LOCK_HIERARCHY_WRITE 32
+#define CFT_LOCK_HIERARCHY (CFT_LOCK_HIERARCHY_READ |
CFT_LOCK_HIERARCHY_WRITE)
+
+/*
+ * CFT_LOCK_CGL_(READ|WRITE): This operation is called with
+ * cgroup_lock() held; it will not run concurrently with any of the
+ * above operations in any cgroup/hierarchy. This should be considered
+ * to be the BKL of cgroups - it should be avoided if you can use
+ * finer-grained locking
+ */
+#define CFT_LOCK_CGL_READ 64
+#define CFT_LOCK_CGL_WRITE 128
+#define CFT_LOCK_CGL (CFT_LOCK_CGL_READ | CFT_LOCK_CGL_WRITE)
+
+#define CFT_LOCK_FOR_READ (CFT_LOCK_ATTACH_READ |  \
+      CFT_LOCK_RMDIR_READ |  \
+      CFT_LOCK_HIERARCHY_READ |  \
+      CFT_LOCK_CGL_READ)
+
+#define CFT_LOCK_FOR_WRITE (CFT_LOCK_ATTACH_WRITE | \
+       CFT_LOCK_RMDIR_WRITE | \
+       CFT_LOCK_HIERARCHY_WRITE | \
+       CFT_LOCK_CGL_WRITE)
+
 struct cftype {
  /* By convention, the name should begin with the name of the
   * subsystem, followed by a period */
  char name[MAX_CFTYPE_NAME];
  int private;
+
+ /*
+  * Determine what locks (if any) are held across calls to
+  * read_X/write_X callback. See lockmode definitions above
+  */
+ int lockmode;
+
```

```
  int (*open) (struct inode *inode, struct file *file);
  ssize_t (*read) (struct cgroup *cgrp, struct cftype *cft,
     struct file *file,
Index: cgroup-2.6.25-mm1/kernel/cgroup.c
===================================================================
--- cgroup-2.6.25-mm1.orig/kernel/cgroup.c
+++ cgroup-2.6.25-mm1/kernel/cgroup.c
@@ -1327,38 +1327,65 @@ enum cgroup_filetype {
  FILE_RELEASE_AGENT,
 };

-static ssize_t cgroup_write_X64(struct cgroup *cgrp, struct cftype *cft,
-    struct file *file,
-    const char __user *userbuf,
-    size_t nbytes, loff_t *unused_ppos)
+
+
+/**
+ * cgroup_file_lock(). Helper for cgroup read/write methods.
+ * @cgrp:  the cgroup being acted on
+ * @cft:   the control file being written to or read from
+ * *write: true if the access is a write access.
+ *
+ * Takes any necessary locks as requested by the control file's
+ * 'lockmode' field; checks (after locking if necessary) that the
+ * control group is not in the process of being destroyed.
+ *
+ * Currently all the locking options are implemented in the same way,
+ * by taking cgroup_mutex. Future patches will add finer-grained
+ * locking.
+ *
+ * Calls to cgroup_file_lock() should always be paired with calls to
+ * cgroup_file_unlock(), even if cgroup_file_lock() returns an error.
+ */
+
+static int cgroup_file_lock(struct cgroup *cgrp, struct cftype *cft, int write)
 {
- char buffer[64];
- int retval = 0;
- char *end;
+ int mask = write ? CFT_LOCK_FOR_WRITE : CFT_LOCK_FOR_READ;
+ BUILD_BUG_ON(CFT_LOCK_FOR_READ != (CFT_LOCK_FOR_WRITE >> 1));

- if (!nbytes)
-  return -EINVAL;
- if (nbytes >= sizeof(buffer))
-  return -E2BIG;
- if (copy_from_user(buffer, userbuf, nbytes))
```

```
-  return -EFAULT;
+ if (cft->lockmode & mask)
+  mutex_lock(&cgroup_mutex);
+ if (cgroup_is_removed(cgrp))
+  return -ENODEV;
+ return 0;
+}
+
+/**
+ * cgroup_file_unlock(): undoes the effect of cgroup_file_lock()
+ */
+
+static void cgroup_file_unlock(struct cgroup *cgrp, struct cftype *cft,
+         int write)
+{
+ int mask = write ? CFT_LOCK_FOR_WRITE : CFT_LOCK_FOR_READ;
+ if (cft->lockmode & mask)
+  mutex_unlock(&cgroup_mutex);
+}

- buffer[nbytes] = 0;     /* nul-terminate */
- strstrip(buffer);
+static ssize_t cgroup_write_X64(struct cgroup *cgrp, struct cftype *cft,
+    const char *buffer)
+{
+ char *end;
  if (cft->write_u64) {
   u64 val = simple_strtoull(buffer, &end, 0);
   if (*end)
    return -EINVAL;
-  retval = cft->write_u64(cgrp, cft, val);
+  return cft->write_u64(cgrp, cft, val);
  } else {
   s64 val = simple_strtoll(buffer, &end, 0);
   if (*end)
    return -EINVAL;
-  retval = cft->write_s64(cgrp, cft, val);
+  return cft->write_s64(cgrp, cft, val);
  }
- if (!retval)
-  retval = nbytes;
- return retval;
 }

 static ssize_t cgroup_common_file_write(struct cgroup *cgrp,
@@ -1426,47 +1453,82 @@ out1:
  return retval;
 }
```

```
-static ssize_t cgroup_file_write(struct file *file, const char __user *buf,
+static ssize_t cgroup_file_write(struct file *file, const char __user *userbuf,
       size_t nbytes, loff_t *ppos)
 {
  struct cftype *cft = __d_cft(file->f_dentry);
  struct cgroup *cgrp = __d_cgrp(file->f_dentry->d_parent);
-
- if (!cft || cgroup_is_removed(cgrp))
-  return -ENODEV;
- if (cft->write)
-  return cft->write(cgrp, cft, file, buf, nbytes, ppos);
- if (cft->write_u64 || cft->write_s64)
-  return cgroup_write_X64(cgrp, cft, file, buf, nbytes, ppos);
- if (cft->trigger) {
-  int ret = cft->trigger(cgrp, (unsigned int)cft->private);
-  return ret ? ret : nbytes;
+ ssize_t retval;
+ char static_buffer[64];
+ char *buffer = static_buffer;
+ ssize_t max_bytes = sizeof(static_buffer) - 1;
+ if (!cft->write && !cft->trigger) {
+  if (!nbytes)
+   return -EINVAL;
+  if (nbytes >= max_bytes)
+   return -E2BIG;
+  if (nbytes >= sizeof(static_buffer)) {
+   /* +1 for nul-terminator */
+   buffer = kmalloc(nbytes + 1, GFP_KERNEL);
+   if (buffer == NULL)
+    return -ENOMEM;
+  }
+  if (copy_from_user(buffer, userbuf, nbytes)) {
+   retval = -EFAULT;
+   goto out_free;
+  }
+  buffer[nbytes] = 0; /* nul-terminate */
+  strstrip(buffer); /* strip -just- trailing whitespace */
  }
- return -EINVAL;
-}

-static ssize_t cgroup_read_u64(struct cgroup *cgrp, struct cftype *cft,
-        struct file *file,
-        char __user *buf, size_t nbytes,
-        loff_t *ppos)
-{
- char tmp[64];
```

```
- u64 val = cft->read_u64(cgrp, cft);
- int len = sprintf(tmp, "%llu\n", (unsigned long long) val);
+ retval = cgroup_file_lock(cgrp, cft, 1);
+ if (retval)
+  goto out_unlock;

- return simple_read_from_buffer(buf, nbytes, ppos, tmp, len);
+ if (cft->write)
+  retval = cft->write(cgrp, cft, file, userbuf, nbytes, ppos);
+ else if (cft->write_u64 || cft->write_s64)
+  retval = cgroup_write_X64(cgrp, cft, buffer);
+ else if (cft->trigger)
+  retval = cft->trigger(cgrp, (unsigned int)cft->private);
+ else
+  retval = -EINVAL;
+ if (retval == 0)
+  retval = nbytes;
+ out_unlock:
+ cgroup_file_unlock(cgrp, cft, 1);
+ out_free:
+ if (buffer != static_buffer)
+  kfree(buffer);
+ return retval;
 }

-static ssize_t cgroup_read_s64(struct cgroup *cgrp, struct cftype *cft,
+static ssize_t cgroup_read_X64(struct cgroup *cgrp, struct cftype *cft,
          struct file *file,
          char __user *buf, size_t nbytes,
          loff_t *ppos)
 {
  char tmp[64];
- s64 val = cft->read_s64(cgrp, cft);
- int len = sprintf(tmp, "%lld\n", (long long) val);
+ ssize_t retval = 0;
+ int len = 0;
+
+ retval = cgroup_file_lock(cgrp, cft, 0);
+ if (retval)
+  goto out_unlock;
+
+ if (cft->read_u64) {
+  u64 val = cft->read_u64(cgrp, cft);
+  len = sprintf(tmp, "%llu\n", (unsigned long long) val);
+ } else {
+  s64 val = cft->read_s64(cgrp, cft);
+  len = sprintf(tmp, "%lld\n", (long long) val);
+ }
```

```
- return simple_read_from_buffer(buf, nbytes, ppos, tmp, len);
+ out_unlock:
+ cgroup_file_unlock(cgrp, cft, 0);
+ if (!retval)
+  retval = simple_read_from_buffer(buf, nbytes, ppos, tmp, len);
+ return retval;
 }

 static ssize_t cgroup_common_file_read(struct cgroup *cgrp,
@@ -1518,16 +1580,21 @@ static ssize_t cgroup_file_read(struct f
  struct cftype *cft = __d_cft(file->f_dentry);
  struct cgroup *cgrp = __d_cgrp(file->f_dentry->d_parent);

- if (!cft || cgroup_is_removed(cgrp))
+ if (cgroup_is_removed(cgrp))
   return -ENODEV;

- if (cft->read)
-  return cft->read(cgrp, cft, file, buf, nbytes, ppos);
- if (cft->read_u64)
-  return cgroup_read_u64(cgrp, cft, file, buf, nbytes, ppos);
- if (cft->read_s64)
-  return cgroup_read_s64(cgrp, cft, file, buf, nbytes, ppos);
- return -EINVAL;
+ if (cft->read) {
+  /* Raw read function - no extra processing by cgroups */
+  ssize_t retval = cgroup_file_lock(cgrp, cft, 0);
+  if (!retval)
+   retval = cft->read(cgrp, cft, file, buf, nbytes, ppos);
+  cgroup_file_unlock(cgrp, cft, 0);
+  return retval;
+ }
+ if (cft->read_u64 || cft->read_s64)
+  return cgroup_read_X64(cgrp, cft, file, buf, nbytes, ppos);
+ else
+  return -EINVAL;
 }

 /*
@@ -1549,15 +1616,28 @@ static int cgroup_map_add(struct cgroup_
 static int cgroup_seqfile_show(struct seq_file *m, void *arg)
 {
  struct cgroup_seqfile_state *state = m->private;
+ struct cgroup *cgrp = state->cgroup;
  struct cftype *cft = state->cft;
+ int retval;
+
```

```
+ retval = cgroup_file_lock(cgrp, cft, 0);
+ if (retval)
+  goto out_unlock;
+
  if (cft->read_map) {
   struct cgroup_map_cb cb = {
    .fill = cgroup_map_add,
    .state = m,
   };
-  return cft->read_map(state->cgroup, cft, &cb);
+  retval = cft->read_map(cgrp, cft, &cb);
+ } else if (cft->read_seq_string) {
+  retval = cft->read_seq_string(cgrp, cft, m);
+ } else {
+  retval = -EINVAL;
  }
- return cft->read_seq_string(state->cgroup, cft, m);
+ out_unlock:
+ cgroup_file_unlock(cgrp, cft, 0);
+ return retval;
 }

 int cgroup_seqfile_release(struct inode *inode, struct file *file)

--

_____
Containers mailing list
Containers@lists.linux-foundation.org
https://lists.linux-foundation.org/mailman/listinfo/containers
```

Subject: [RFC/PATCH 2/8]: CGroup Files: Add a cgroup write_string control file method

Posted by Paul Menage on Tue, 13 May 2008 06:37:09 GMT

View Forum Message <> Reply to Message

This patch adds a write_string() method for cgroups control files. The semantics are that a buffer is copied from userspace to kernelspace and the handler function invoked on that buffer. Any control group locking is done after the copy from userspace has occurred. The buffer is guaranteed to be nul-terminated, and no longer than max_write_len (defaulting to 64 bytes if unspecified). Later patches will convert existing raw file write handlers in control group subsystems to use this method.

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

```
 include/linux/cgroup.h |   10 ++++++++++
 kernel/cgroup.c        |    5 ++++-
 2 files changed, 14 insertions(+), 1 deletion(-)

Index: cgroup-2.6.25-mm1/include/linux/cgroup.h
===================================================================
--- cgroup-2.6.25-mm1.orig/include/linux/cgroup.h
+++ cgroup-2.6.25-mm1/include/linux/cgroup.h
@@ -281,6 +281,10 @@ struct cftype {
   */
  int lockmode;

+ /* If non-zero, defines the maximum length of string that can
+  * be passed to write_string; defaults to 64 */
+ int max_write_len;
+
  int (*open) (struct inode *inode, struct file *file);
  ssize_t (*read) (struct cgroup *cgrp, struct cftype *cft,
     struct file *file,
@@ -323,6 +327,12 @@ struct cftype {
   * write_s64() is a signed version of write_u64()
   */
  int (*write_s64) (struct cgroup *cgrp, struct cftype *cft, s64 val);
+ /*
+  * write_string() is passed a nul-terminated kernelspace
+  * buffer of maximum length determined by max_write_len
+  */
+ int (*write_string) (struct cgroup *cgrp, struct cftype *cft,
+        char *buffer);

  /*
   * trigger() callback can be used to get some kick from the
Index: cgroup-2.6.25-mm1/kernel/cgroup.c
===================================================================
--- cgroup-2.6.25-mm1.orig/kernel/cgroup.c
+++ cgroup-2.6.25-mm1/kernel/cgroup.c
@@ -1461,7 +1461,7 @@ static ssize_t cgroup_file_write(struct
  ssize_t retval;
  char static_buffer[64];
  char *buffer = static_buffer;
- ssize_t max_bytes = sizeof(static_buffer) - 1;
+ ssize_t max_bytes =  cft->max_write_len ?: sizeof(static_buffer) - 1;
  if (!cft->write && !cft->trigger) {
   if (!nbytes)
    return -EINVAL;
@@ -1489,6 +1489,8 @@ static ssize_t cgroup_file_write(struct
   retval = cft->write(cgrp, cft, file, userbuf, nbytes, ppos);
  else if (cft->write_u64 || cft->write_s64)
```

```
  retval = cgroup_write_X64(cgrp, cft, buffer);
+ else if (cft->write_string)
+  retval = cft->write_string(cgrp, cft, buffer);
 else if (cft->trigger)
  retval = cft->trigger(cgrp, (unsigned int)cft->private);
 else
@@ -1651,6 +1653,7 @@ static struct file_operations cgroup_seq
 .read = seq_read,
 .llseek = seq_lseek,
 .release = cgroup_seqfile_release,
+ .write = cgroup_file_write,
};

static int cgroup_file_open(struct inode *inode, struct file *file)
```

--

____
Containers mailing list
Containers@lists.linux-foundation.org
https://lists.linux-foundation.org/mailman/listinfo/containers

---

Subject: [RFC/PATCH 3/8]: CGroup Files: Move the release_agent file to use typed handlers
Posted by Paul Menage on Tue, 13 May 2008 06:37:10 GMT
View Forum Message <> Reply to Message

Adds cgroup_release_agent_write() and cgroup_release_agent_show()
methods to handle writing/reading the path to a cgroup hierarchy's
release agent. As a result, cgroup_common_file_read() is now unnecessary.

As part of the change, a previously-tolerated race in
cgroup_release_agent() is avoided by copying the current
release_agent_path prior to calling call_usermode_helper().

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

```
---
 kernel/cgroup.c |  103 +++++++++++++++++-----------------------------------
 1 file changed, 35 insertions(+), 68 deletions(-)

Index: cgroup-2.6.25-mm1/kernel/cgroup.c
===================================================================
--- cgroup-2.6.25-mm1.orig/kernel/cgroup.c
+++ cgroup-2.6.25-mm1/kernel/cgroup.c
@@ -89,11 +89,7 @@ struct cgroupfs_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 child cgroups exist, you could miss a
-  * notification. We ensure that it's always a valid
-  * NUL-terminated string */
+ /* The path to use for release notifications. */
  char release_agent_path[PATH_MAX];
 };

@@ -1371,6 +1367,22 @@ static void cgroup_file_unlock(struct cg
   mutex_unlock(&cgroup_mutex);
 }

+static int cgroup_release_agent_write(struct cgroup *cgrp, struct cftype *cft,
+          char *buffer)
+{
+ BUILD_BUG_ON(sizeof(cgrp->root->release_agent_path) < PATH_MAX);
+ strcpy(cgrp->root->release_agent_path, buffer);
+ return 0;
+}
+
+static int cgroup_release_agent_show(struct cgroup *cgrp, struct cftype *cft,
+         struct seq_file *seq)
+{
+ seq_puts(seq, cgrp->root->release_agent_path);
+ seq_putc(seq, '\n');
+ return 0;
+}
+
 static ssize_t cgroup_write_X64(struct cgroup *cgrp, struct cftype *cft,
     const char *buffer)
 {
@@ -1435,10 +1447,6 @@ static ssize_t cgroup_common_file_write(
   else
    clear_bit(CGRP_NOTIFY_ON_RELEASE, &cgrp->flags);
   break;
- case FILE_RELEASE_AGENT:
-  BUILD_BUG_ON(sizeof(cgrp->root->release_agent_path) < PATH_MAX);
-  strcpy(cgrp->root->release_agent_path, buffer);
-  break;
  default:
   retval = -EINVAL;
   goto out2;
@@ -1533,49 +1541,6 @@ static ssize_t cgroup_read_X64(struct cg
  return retval;
 }
```

```
-static ssize_t cgroup_common_file_read(struct cgroup *cgrp,
-       struct cftype *cft,
-       struct file *file,
-       char __user *buf,
-       size_t nbytes, loff_t *ppos)
-{
- enum cgroup_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 cgroupfs_root *root;
-  size_t n;
-  mutex_lock(&cgroup_mutex);
-  root = cgrp->root;
-  n = strnlen(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(&cgroup_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 cgroup_file_read(struct file *file, char __user *buf,
        size_t nbytes, loff_t *ppos)
 {
@@ -2333,8 +2298,10 @@ static struct cftype files[] = {
```

```
 static struct cftype cft_release_agent = {
  .name = "release_agent",
- .read = cgroup_common_file_read,
- .write = cgroup_common_file_write,
+ .read_seq_string = cgroup_release_agent_show,
+ .write_string = cgroup_release_agent_write,
+ .lockmode = CFT_LOCK_CGL,
+ .max_write_len = PATH_MAX,
  .private = FILE_RELEASE_AGENT,
 };

@@ -3161,27 +3128,25 @@ static void cgroup_release_agent(struct
  while (!list_empty(&release_list)) {
   char *argv[3], *envp[3];
   int i;
-  char *pathbuf;
+  char *pathbuf = NULL, *agentbuf = NULL;
   struct cgroup *cgrp = list_entry(release_list.next,
         struct cgroup,
         release_list);
   list_del_init(&cgrp->release_list);
   spin_unlock(&release_list_lock);
   pathbuf = kmalloc(PAGE_SIZE, GFP_KERNEL);
-  if (!pathbuf) {
-   spin_lock(&release_list_lock);
-   continue;
-  }
-
-  if (cgroup_path(cgrp, pathbuf, PAGE_SIZE) < 0) {
-   kfree(pathbuf);
-   spin_lock(&release_list_lock);
-   continue;
-  }
+  if (!pathbuf)
+   goto continue_free;
+  if (cgroup_path(cgrp, pathbuf, PAGE_SIZE) < 0)
+   goto continue_free;
+  agentbuf = kmalloc(PATH_MAX, GFP_KERNEL);
+  if (!agentbuf)
+   goto continue_free;
+  strcpy(agentbuf, cgrp->root->release_agent_path);

   i = 0;
-  argv[i++] = cgrp->root->release_agent_path;
-  argv[i++] = (char *)pathbuf;
+  argv[i++] = agentbuf;
+  argv[i++] = pathbuf;
   argv[i] = NULL;
```

```
   i = 0;
@@ -3195,8 +3160,10 @@ static void cgroup_release_agent(struct
    * be a slow process */
   mutex_unlock(&cgroup_mutex);
   call_usermodehelper(argv[0], argv, envp, UMH_WAIT_EXEC);
-  kfree(pathbuf);
   mutex_lock(&cgroup_mutex);
+ continue_free:
+  kfree(pathbuf);
+  kfree(agentbuf);
   spin_lock(&release_list_lock);
  }
  spin_unlock(&release_list_lock);
```

--

Subject: [RFC/PATCH 4/8]: CGroup Files: Move notify_on_release file to separate write handler

Posted by Paul Menage on Tue, 13 May 2008 06:37:11 GMT

View Forum Message <> Reply to Message

This patch moves the write handler for the cgroups notify_on_release file into a separate handler. This handler requires no cgroups locking since it relies on atomic bitops for synchronization.

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

---
 kernel/cgroup.c | 27 +++++++++++++++++----------
 1 file changed, 15 insertions(+), 12 deletions(-)

Index: cgroup-2.6.25-mm1/kernel/cgroup.c
```
===================================================================
--- cgroup-2.6.25-mm1.orig/kernel/cgroup.c
+++ cgroup-2.6.25-mm1/kernel/cgroup.c
@@ -1440,13 +1440,6 @@ static ssize_t cgroup_common_file_write(
  case FILE_TASKLIST:
   retval = attach_task_by_pid(cgrp, buffer);
   break;
- case FILE_NOTIFY_ON_RELEASE:
-  clear_bit(CGRP_RELEASABLE, &cgrp->flags);
-  if (simple_strtoul(buffer, NULL, 10) != 0)
```

```
-   set_bit(CGRP_NOTIFY_ON_RELEASE, &cgrp->flags);
-  else
-   clear_bit(CGRP_NOTIFY_ON_RELEASE, &cgrp->flags);
-  break;
   default:
    retval = -EINVAL;
    goto out2;
@@ -2275,6 +2268,18 @@ static u64 cgroup_read_notify_on_release
  return notify_on_release(cgrp);
 }

+static int cgroup_write_notify_on_release(struct cgroup *cgrp,
+       struct cftype *cft,
+       u64 val)
+{
+ clear_bit(CGRP_RELEASABLE, &cgrp->flags);
+ if (val)
+  set_bit(CGRP_NOTIFY_ON_RELEASE, &cgrp->flags);
+ else
+  clear_bit(CGRP_NOTIFY_ON_RELEASE, &cgrp->flags);
+ return 0;
+}
+
 /*
  * for the common functions, 'private' gives the type of file
  */
@@ -2291,7 +2296,7 @@ static struct cftype files[] = {
    .name = "notify_on_release",
    .read_u64 = cgroup_read_notify_on_release,
-   .write = cgroup_common_file_write,
+   .write_u64 = cgroup_write_notify_on_release,
    .private = FILE_NOTIFY_ON_RELEASE,
   },
 };

--
```


Subject: [RFC/PATCH 5/8]: CGroup Files: Turn attach_task_by_pid directly into a cgroup write handler
Posted by Paul Menage on Tue, 13 May 2008 06:37:12 GMT
View Forum Message <> Reply to Message

This patch changes attach_task_by_pid() to take a u64 rather than a
string; as a result it can be called directly as a control groups
write_u64 handler, and cgroup_common_file_write() can be removed.

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

---
 kernel/cgroup.c | 67 ++--
 1 file changed, 3 insertions(+), 64 deletions(-)

Index: cgroup-2.6.25-mm1/kernel/cgroup.c
===
--- cgroup-2.6.25-mm1.orig/kernel/cgroup.c
+++ cgroup-2.6.25-mm1/kernel/cgroup.c
@@ -503,10 +503,6 @@ static struct css_set *find_css_set(
  * knows that the cgroup won't be removed, as cgroup_rmdir()
  * needs that mutex.
  *
- * The cgroup_common_file_write handler for operations that modify
- * the cgroup hierarchy holds cgroup_mutex across the entire operation,
- * single threading all such cgroup modifications across the system.
- *
  * The fork and exit callbacks cgroup_fork() and cgroup_exit(), don't
  * (usually) take cgroup_mutex. These are the two most performance
  * critical pieces of code here. The exception occurs on cgroup_exit(),
@@ -1280,15 +1276,11 @@ int cgroup_attach_task(struct cgroup *cg
  * Attach task with pid 'pid' to cgroup 'cgrp'. Call with
  * cgroup_mutex, may take task_lock of task
  */
-static int attach_task_by_pid(struct cgroup *cgrp, char *pidbuf)
+static int attach_task_by_pid(struct cgroup *cgrp, struct cftype *cft, u64 pid)
 {
- pid_t pid;
  struct task_struct *tsk;
  int ret;

- if (sscanf(pidbuf, "%d", &pid) != 1)
- return -EIO;
-
  if (pid) {
   rcu_read_lock();
   tsk = find_task_by_vpid(pid);
@@ -1400,60 +1392,6 @@ static ssize_t cgroup_write_X64(struct c
  }
 }

-static ssize_t cgroup_common_file_write(struct cgroup *cgrp,
-       struct cftype *cft,

```
-        struct file *file,
-        const char __user *userbuf,
-        size_t nbytes, loff_t *unused_ppos)
-{
- enum cgroup_filetype type = cft->private;
- char *buffer;
- int retval = 0;
-
- if (nbytes >= PATH_MAX)
-  return -E2BIG;
-
- /* +1 for nul-terminator */
- buffer = kmalloc(nbytes + 1, GFP_KERNEL);
- if (buffer == NULL)
-  return -ENOMEM;
-
- if (copy_from_user(buffer, userbuf, nbytes)) {
-  retval = -EFAULT;
-  goto out1;
- }
- buffer[nbytes] = 0; /* nul-terminate */
- strstrip(buffer); /* strip -just- trailing whitespace */
-
- mutex_lock(&cgroup_mutex);
-
- /*
-  * This was already checked for in cgroup_file_write(), but
-  * check again now we're holding cgroup_mutex.
-  */
- if (cgroup_is_removed(cgrp)) {
-  retval = -ENODEV;
-  goto out2;
- }
-
- switch (type) {
- case FILE_TASKLIST:
-  retval = attach_task_by_pid(cgrp, buffer);
-  break;
- default:
-  retval = -EINVAL;
-  goto out2;
- }
-
- if (retval == 0)
-  retval = nbytes;
-out2:
- mutex_unlock(&cgroup_mutex);
-out1:
```

```
- kfree(buffer);
- return retval;
-}
-
 static ssize_t cgroup_file_write(struct file *file, const char __user *userbuf,
       size_t nbytes, loff_t *ppos)
 {
@@ -2288,8 +2226,9 @@ static struct cftype files[] = {
   .name = "tasks",
   .open = cgroup_tasks_open,
   .read = cgroup_tasks_read,
-  .write = cgroup_common_file_write,
+  .write_u64 = attach_task_by_pid,
   .release = cgroup_tasks_release,
+  .lockmode = CFT_LOCK_ATTACH,
   .private = FILE_TASKLIST,
  },
```

--

_____

Containers mailing list
Containers@lists.linux-foundation.org
https://lists.linux-foundation.org/mailman/listinfo/containers

---

Subject: [RFC/PATCH 6/8]: CGroup Files: Remove cpuset_common_file_write()

Posted by Paul Menage on Tue, 13 May 2008 06:37:13 GMT

View Forum Message <> Reply to Message

This patch tweaks the signatures of the update_cpumask() and
update_nodemask() functions so that they can be called directly as
handlers for the new cgroups write_string() method.

This allows cpuset_common_file_write() to be removed.

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

---
 kernel/cpuset.c | 73 +++++++--
 1 file changed, 10 insertions(+), 63 deletions(-)

Index: cgroup-2.6.25-mm1/kernel/cpuset.c
===
--- cgroup-2.6.25-mm1.orig/kernel/cpuset.c
+++ cgroup-2.6.25-mm1/kernel/cpuset.c
@@ -224,10 +224,6 @@ static struct cpuset top_cpuset = {
  * The task_struct fields mems_allowed and mems_generation may only

```
  * be accessed in the context of that task, so require no locks.
  *
- * The cpuset_common_file_write handler for operations that modify
- * the cpuset hierarchy holds cgroup_mutex across the entire operation,
- * single threading all such cpuset modifications across the system.
- *
  * The cpuset_common_file_read() handlers only hold callback_mutex across
  * small pieces of code, such as when reading out possibly multi-word
  * cpumasks and nodemasks.
@@ -747,8 +743,9 @@ static void cpuset_change_cpumask(struct
  * @cs: the cpuset to consider
  * @buf: buffer of cpu numbers written to this cpuset
  */
-static int update_cpumask(struct cpuset *cs, char *buf)
+static int update_cpumask(struct cgroup *cgrp, struct cftype *cft, char *buf)
 {
+ struct cpuset *cs = cgroup_cs(cgrp);
  struct cpuset trialcs;
  struct cgroup_scanner scan;
  struct ptr_heap heap;
@@ -767,7 +764,6 @@ static int update_cpumask(struct cpuset
   * that parsing.  The validate_change() call ensures that cpusets
   * with tasks have cpus.
   */
- buf = strstrip(buf);
  if (!*buf) {
   cpus_clear(trialcs.cpus_allowed);
  } else {
@@ -875,8 +871,9 @@ static void cpuset_migrate_mm(struct mm_

 static void *cpuset_being_rebound;

-static int update_nodemask(struct cpuset *cs, char *buf)
+static int update_nodemask(struct cgroup *cgrp, struct cftype *cft, char *buf)
 {
+ struct cpuset *cs = cgroup_cs(cgrp);
  struct cpuset trialcs;
  nodemask_t oldmem;
  struct task_struct *p;
@@ -902,7 +899,6 @@ static int update_nodemask(struct cpuset
   * that parsing.  The validate_change() call ensures that cpusets
   * with tasks have memory.
   */
- buf = strstrip(buf);
  if (!*buf) {
   nodes_clear(trialcs.mems_allowed);
  } else {
@@ -1201,59 +1197,6 @@ typedef enum {
```

```
  FILE_SPREAD_SLAB,
 } cpuset_filetype_t;

-static ssize_t cpuset_common_file_write(struct cgroup *cont,
-     struct cftype *cft,
-     struct file *file,
-     const char __user *userbuf,
-     size_t nbytes, loff_t *unused_ppos)
-{
- struct cpuset *cs = cgroup_cs(cont);
- cpuset_filetype_t type = cft->private;
- char *buffer;
- int retval = 0;
-
- /* Crude upper limit on largest legitimate cpulist user might write. */
- if (nbytes > 100U + 6 * max(NR_CPUS, MAX_NUMNODES))
-  return -E2BIG;
-
- /* +1 for nul-terminator */
- if ((buffer = kmalloc(nbytes + 1, GFP_KERNEL)) == 0)
-  return -ENOMEM;
-
- if (copy_from_user(buffer, userbuf, nbytes)) {
-  retval = -EFAULT;
-  goto out1;
- }
- buffer[nbytes] = 0; /* nul-terminate */
-
- cgroup_lock();
-
- if (cgroup_is_removed(cont)) {
-  retval = -ENODEV;
-  goto out2;
- }
-
- switch (type) {
- case FILE_CPULIST:
-  retval = update_cpumask(cs, buffer);
-  break;
- case FILE_MEMLIST:
-  retval = update_nodemask(cs, buffer);
-  break;
- default:
-  retval = -EINVAL;
-  goto out2;
- }
-
- if (retval == 0)
```

```
-  retval = nbytes;
-out2:
- cgroup_unlock();
-out1:
- kfree(buffer);
- return retval;
-}
-
 static int cpuset_write_u64(struct cgroup *cgrp, struct cftype *cft, u64 val)
 {
  int retval = 0;
@@ -1412,14 +1355,18 @@ static struct cftype files[] = {
  {
   .name = "cpus",
   .read = cpuset_common_file_read,
-  .write = cpuset_common_file_write,
+  .write_string = update_cpumask,
+  .max_write_len = (100U + 6 * NR_CPUS),
+  .lockmode = CFT_LOCK_CGL_WRITE,
   .private = FILE_CPULIST,
  },

  {
   .name = "mems",
   .read = cpuset_common_file_read,
-  .write = cpuset_common_file_write,
+  .write_string = update_nodemask,
+  .max_write_len = (100U + 6 * MAX_NUMNODES),
+  .lockmode = CFT_LOCK_CGL_WRITE,
   .private = FILE_MEMLIST,
  },
```

--

Subject: [RFC/PATCH 7/8]: CGroup Files: Convert devcgroup_access_write() into a cgroup write_string() handler

Posted by Paul Menage on Tue, 13 May 2008 06:37:14 GMT

View Forum Message <> Reply to Message

This patch converts devcgroup_access_write() from a raw file handler into a handler for the cgroup write_string() method. This allows some boilerplate copying/locking/checking to be removed and simplifies the

cleanup path, since these functions are performed by the cgroups
framework before calling the handler.

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

---
 security/device_cgroup.c | 79 +++++++++++++++++++++++++---------------------
 1 file changed, 22 insertions(+), 57 deletions(-)

Index: cgroup-2.6.25-mm1/security/device_cgroup.c
===
--- cgroup-2.6.25-mm1.orig/security/device_cgroup.c
+++ cgroup-2.6.25-mm1/security/device_cgroup.c
@@ -323,14 +323,14 @@ static int parent_has_perm(struct cgroup
  * new access is only allowed if you're in the top-level cgroup, or your
  * parent cgroup has the access you're asking for.
  */
-static ssize_t devcgroup_access_write(struct cgroup *cgroup, struct cftype *cft,
-    struct file *file, const char __user *userbuf,
-    size_t nbytes, loff_t *ppos)
+static int devcgroup_access_write(struct cgroup *cgroup,
+   struct cftype *cft,
+   char *buffer)
 {
  struct cgroup *cur_cgroup;
  struct dev_cgroup *devcgroup, *cur_devcgroup;
  int filetype = cft->private;
- char *buffer, *b;
+ char *b;
  int retval = 0, count;
  struct dev_whitelist_item wh;

@@ -341,22 +341,6 @@ static ssize_t devcgroup_access_write(st
  cur_cgroup = task_cgroup(current, devices_subsys.subsys_id);
  cur_devcgroup = cgroup_to_devcgroup(cur_cgroup);

- buffer = kmalloc(nbytes+1, GFP_KERNEL);
- if (!buffer)
- return -ENOMEM;
-
- if (copy_from_user(buffer, userbuf, nbytes)) {
- retval = -EFAULT;
- goto out1;
- }
- buffer[nbytes] = 0; /* nul-terminate */
-
- cgroup_lock();
- if (cgroup_is_removed(cgroup)) {

```
-  retval = -ENODEV;
-  goto out2;
- }
-
  memset(&wh, 0, sizeof(wh));
  b = buffer;

@@ -372,14 +356,11 @@ static ssize_t devcgroup_access_write(st
   wh.type = DEV_CHAR;
   break;
  default:
-  retval = -EINVAL;
-  goto out2;
+  return -EINVAL;
  }
  b++;
- if (!isspace(*b)) {
-  retval = -EINVAL;
-  goto out2;
- }
+ if (!isspace(*b))
+  return -EINVAL;
  b++;
  if (*b == '*') {
   wh.major = ~0;
@@ -391,13 +372,10 @@ static ssize_t devcgroup_access_write(st
    b++;
   }
  } else {
-  retval = -EINVAL;
-  goto out2;
- }
- if (*b != ':') {
-  retval = -EINVAL;
-  goto out2;
+  return -EINVAL;
  }
+ if (*b != ':')
+  return -EINVAL;
  b++;

  /* read minor */
@@ -411,13 +389,10 @@ static ssize_t devcgroup_access_write(st
    b++;
   }
  } else {
-  retval = -EINVAL;
-  goto out2;
```

```
- }
- if (!isspace(*b)) {
-  retval = -EINVAL;
-  goto out2;
+  return -EINVAL;
  }
+ if (!isspace(*b))
+  return -EINVAL;
  for (b++, count = 0; count < 3; count++, b++) {
   switch (*b) {
   case 'r':
@@ -434,8 +409,7 @@ static ssize_t devcgroup_access_write(st
    count = 3;
    break;
   default:
-   retval = -EINVAL;
-   goto out2;
+   return -EINVAL;
   }
  }

@@ -444,38 +418,29 @@ handle:
  switch (filetype) {
  case DEVCG_ALLOW:
   if (!parent_has_perm(cgroup, &wh))
-   retval = -EPERM;
-  else
-   retval = dev_whitelist_add(devcgroup, &wh);
-  break;
+   return -EPERM;
+  return dev_whitelist_add(devcgroup, &wh);
  case DEVCG_DENY:
   dev_whitelist_rm(devcgroup, &wh);
   break;
  default:
-  retval = -EINVAL;
-  goto out2;
+  return -EINVAL;
  }
-
- if (retval == 0)
-  retval = nbytes;
-
-out2:
- cgroup_unlock();
-out1:
- kfree(buffer);
- return retval;
```

```
+ return 0;
 }

 static struct cftype dev_cgroup_files[] = {
  {
   .name = "allow",
-  .write  = devcgroup_access_write,
+  .write_string  = devcgroup_access_write,
   .private = DEVCG_ALLOW,
+  .lockmode = CFT_LOCK_CGL_WRITE,
  },
  {
   .name = "deny",
-  .write = devcgroup_access_write,
+  .write_string = devcgroup_access_write,
   .private = DEVCG_DENY,
+  .lockmode = CFT_LOCK_CGL_WRITE,
  },
  {
   .name = "list",

--
```

_____

Containers mailing list
Containers@lists.linux-foundation.org
https://lists.linux-foundation.org/mailman/listinfo/containers

---

Subject: [RFC/PATCH 8/8]: CGroup Files: Convert res_counter_write() to be a cgroups write_string() handler

Posted by Paul Menage on Tue, 13 May 2008 06:37:15 GMT
View Forum Message <> Reply to Message

Currently read_counter_write() is a raw file handler even though it's ultimately taking a number, since in some cases it wants to pre-process the string when converting it to a number.

This patch converts res_counter_write() from a raw file handler to a write_string() handler; this allows some of the boilerplate copying/locking/checking to be removed, and simplies the cleanup path, since these functions are now performed by the cgroups framework.

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

```
---
 include/linux/res_counter.h |    4 ++--
 kernel/res_counter.c        |   36 +++++++++----------------------
 mm/memcontrol.c             |   11 +++++-----
```

```
 3 files changed, 16 insertions(+), 35 deletions(-)

Index: cgroup-2.6.25-mm1/include/linux/res_counter.h
===================================================================
--- cgroup-2.6.25-mm1.orig/include/linux/res_counter.h
+++ cgroup-2.6.25-mm1/include/linux/res_counter.h
@@ -63,8 +63,8 @@ u64 res_counter_read_u64(struct res_coun
 ssize_t res_counter_read(struct res_counter *counter, int member,
   const char __user *buf, size_t nbytes, loff_t *pos,
   int (*read_strategy)(unsigned long long val, char *s));
-ssize_t res_counter_write(struct res_counter *counter, int member,
-  const char __user *buf, size_t nbytes, loff_t *pos,
+int res_counter_write(struct res_counter *counter, int member,
+  char *buffer,
   int (*write_strategy)(char *buf, unsigned long long *val));

 /*
Index: cgroup-2.6.25-mm1/kernel/res_counter.c
===================================================================
--- cgroup-2.6.25-mm1.orig/kernel/res_counter.c
+++ cgroup-2.6.25-mm1/kernel/res_counter.c
@@ -102,44 +102,26 @@ u64 res_counter_read_u64(struct res_coun
  return *res_counter_member(counter, member);
 }

-ssize_t res_counter_write(struct res_counter *counter, int member,
-  const char __user *userbuf, size_t nbytes, loff_t *pos,
-  int (*write_strategy)(char *st_buf, unsigned long long *val))
+int res_counter_write(
+ struct res_counter *counter, int member,
+ char *buf,
+ int (*write_strategy)(char *st_buf, unsigned long long *val))
 {
- int ret;
- char *buf, *end;
+ char *end;
  unsigned long flags;
  unsigned long long tmp, *val;

- buf = kmalloc(nbytes + 1, GFP_KERNEL);
- ret = -ENOMEM;
- if (buf == NULL)
-  goto out;
-
- buf[nbytes] = '\0';
- ret = -EFAULT;
- if (copy_from_user(buf, userbuf, nbytes))
-  goto out_free;
```

```
-
- ret = -EINVAL;
-
- strstrip(buf);
  if (write_strategy) {
-  if (write_strategy(buf, &tmp)) {
-   goto out_free;
-  }
+  if (write_strategy(buf, &tmp))
+   return -EINVAL;
  } else {
   tmp = simple_strtoull(buf, &end, 10);
   if (*end != '\0')
-   goto out_free;
+   return -EINVAL;
  }
  spin_lock_irqsave(&counter->lock, flags);
  val = res_counter_member(counter, member);
  *val = tmp;
  spin_unlock_irqrestore(&counter->lock, flags);
- ret = nbytes;
-out_free:
- kfree(buf);
-out:
- return ret;
+ return 0;
 }
Index: cgroup-2.6.25-mm1/mm/memcontrol.c
===================================================================
--- cgroup-2.6.25-mm1.orig/mm/memcontrol.c
+++ cgroup-2.6.25-mm1/mm/memcontrol.c
@@ -848,13 +848,12 @@ static u64 mem_cgroup_read(struct cgroup
        cft->private);
 }

-static ssize_t mem_cgroup_write(struct cgroup *cont, struct cftype *cft,
-    struct file *file, const char __user *userbuf,
-    size_t nbytes, loff_t *ppos)
+static int mem_cgroup_write(struct cgroup *cont, struct cftype *cft,
+       char *buffer)
 {
  return res_counter_write(&mem_cgroup_from_cont(cont)->res,
-    cft->private, userbuf, nbytes, ppos,
-    mem_cgroup_write_strategy);
+     cft->private, buffer,
+     mem_cgroup_write_strategy);
 }
```

```
 static int mem_cgroup_reset(struct cgroup *cont, unsigned int event)
@@ -929,7 +928,7 @@ static struct cftype mem_cgroup_files[]
  {
   .name = "limit_in_bytes",
   .private = RES_LIMIT,
-  .write = mem_cgroup_write,
+  .write_string = mem_cgroup_write,
   .read_u64 = mem_cgroup_read,
  },
  {

--
```

Subject: Re: [RFC/PATCH 1/8]: CGroup Files: Add locking mode to cgroups control files

Posted by Li Zefan on Tue, 13 May 2008 09:23:29 GMT

View Forum Message <> Reply to Message

menage@google.com wrote:
> Different cgroup files have different stability requirements of the
> cgroups framework while the handler is running; currently most
> subsystems that don't have their own internal synchronization just
> call cgroup_lock()/cgroup_unlock(), which takes the global cgroup_mutex.
>
> This patch introduces a range of locking modes that can be requested
> by a control file; currently these are all implemented internally by
> taking cgroup_mutex, but expressing the intention will make it simpler
> to move to a finer-grained locking scheme in the future.
>
> Signed-off-by: Paul Menage<menage@google.com>
>

This patch series looks good to me. I've reviewed those patches and didn't
see anything wrong, except a little comments below.

[..snap..]

> -static ssize_t cgroup_write_X64(struct cgroup *cgrp, struct cftype *cft,
> -   struct file *file,
> -   const char __user *userbuf,
> -   size_t nbytes, loff_t *unused_ppos)
> +
> +

> +/**
> + * cgroup_file_lock(). Helper for cgroup read/write methods.
> + * @cgrp: the cgroup being acted on
> + * @cft: the control file being written to or read from
> + * *write: true if the access is a write access.

s/*write/@write

[..snip..]

>
> static ssize_t cgroup_common_file_read(struct cgroup *cgrp,
> @@ -1518,16 +1580,21 @@ static ssize_t cgroup_file_read(struct f
>   struct cftype *cft = __d_cft(file->f_dentry);
>   struct cgroup *cgrp = __d_cgrp(file->f_dentry->d_parent);
>
> - if (!cft || cgroup_is_removed(cgrp))
> + if (cgroup_is_removed(cgrp))
>   return -ENODEV;
>

This check seems redundant now.

> - if (cft->read)
> - return cft->read(cgrp, cft, file, buf, nbytes, ppos);
> - if (cft->read_u64)
> - return cgroup_read_u64(cgrp, cft, file, buf, nbytes, ppos);
> - if (cft->read_s64)
> - return cgroup_read_s64(cgrp, cft, file, buf, nbytes, ppos);
> - return -EINVAL;
> + if (cft->read) {
> + /* Raw read function - no extra processing by cgroups */
> + ssize_t retval = cgroup_file_lock(cgrp, cft, 0);
> + if (!retval)
> +   retval = cft->read(cgrp, cft, file, buf, nbytes, ppos);
> + cgroup_file_unlock(cgrp, cft, 0);
> + return retval;
> + }
> + if (cft->read_u64 || cft->read_s64)
> + return cgroup_read_X64(cgrp, cft, file, buf, nbytes, ppos);
> + else
> + return -EINVAL;
> }

Subject: Re: [RFC/PATCH 1/8]: CGroup Files: Add locking mode to cgroups control files
Posted by akpm on Tue, 13 May 2008 20:01:27 GMT
View Forum Message <> Reply to Message

Fear, doubt and resistance!

On Mon, 12 May 2008 23:37:08 -0700
menage@google.com wrote:

> Different cgroup files have different stability requirements of the
> cgroups framework while the handler is running; currently most
> subsystems that don't have their own internal synchronization just
> call cgroup_lock()/cgroup_unlock(), which takes the global cgroup_mutex.
>
> This patch introduces a range of locking modes that can be requested
> by a control file; currently these are all implemented internally by
> taking cgroup_mutex, but expressing the intention will make it simpler
> to move to a finer-grained locking scheme in the future.
>

This, umm, doesn't seem to do much to make the kernel a simpler place.

Do we expect to gain much from this? Hope so... What?

> Index: cgroup-2.6.25-mm1/include/linux/cgroup.h
> ===
> --- cgroup-2.6.25-mm1.orig/include/linux/cgroup.h
> +++ cgroup-2.6.25-mm1/include/linux/cgroup.h
> @@ -200,11 +200,87 @@ struct cgroup_map_cb {
>   */
>
> #define MAX_CFTYPE_NAME 64
> +
> +/* locking modes for control files.
> + *
> + * These determine what level of guarantee the file handler wishes
> + * cgroups to provide about the stability of control group entities
> + * for the duration of the handler callback.
> + *
> + * The minimum guarantee is that the subsystem state for this
> + * subsystem will not be freed (via a call to the subsystem's
> + * destroy() callback) until after the control file handler
> + * returns. This guarantee is provided by the fact that the open
> + * dentry for the control file keeps its parent (cgroup) dentry alive,
> + * which in turn keeps the cgroup object from being actually freed
> + * (although it can be moved into the removed state in the
> + * meantime). This is suitable for subsystems that completely control
> + * their own synchronization.

> + *
> + * Other possible guarantees are given below.
> + *
> + * XXX_READ bits are used for a read operation on the control file,
> + * XXX_WRITE bits are used for a write operation on the control file
> + */

Vague handwaving: lockdep doesn't know anything about any of this.
Whereas if we were more conventional in using separate locks and
suitable lock types for each application, we would retain full lockdep
coverage.

> +/*
> + * CFT_LOCK_ATTACH_(READ|WRITE): This operation will not run
> + * concurrently with a task movement into or out of this cgroup.
> + */
> +#define CFT_LOCK_ATTACH_READ 1
> +#define CFT_LOCK_ATTACH_WRITE 2
> +#define CFT_LOCK_ATTACH (CFT_LOCK_ATTACH_READ | CFT_LOCK_ATTACH_WRITE)
> +
> +/*
> + * CFT_LOCK_RMDIR_(READ|WRITE): This operation will not run
> + * concurrently with the removal of the affected cgroup.
> + */
> +#define CFT_LOCK_RMDIR_READ 4
> +#define CFT_LOCK_RMDIR_WRITE 8
> +#define CFT_LOCK_RMDIR (CFT_LOCK_RMDIR_READ | CFT_LOCK_RMDIR_WRITE)
> +
> +/*
> + * CFT_LOCK_HIERARCHY_(READ|WRITE): This operation will not run
> + * concurrently with a cgroup creation or removal in this hierarchy,
> + * or a bind/move/unbind for this subsystem.
> + */
> +#define CFT_LOCK_HIERARCHY_READ 16
> +#define CFT_LOCK_HIERARCHY_WRITE 32
> +#define CFT_LOCK_HIERARCHY (CFT_LOCK_HIERARCHY_READ |
CFT_LOCK_HIERARCHY_WRITE)
> +
> +/*
> + * CFT_LOCK_CGL_(READ|WRITE): This operation is called with
> + * cgroup_lock() held; it will not run concurrently with any of the
> + * above operations in any cgroup/hierarchy. This should be considered
> + * to be the BKL of cgroups - it should be avoided if you can use
> + * finer-grained locking
> + */
> +#define CFT_LOCK_CGL_READ 64
> +#define CFT_LOCK_CGL_WRITE 128
> +#define CFT_LOCK_CGL (CFT_LOCK_CGL_READ | CFT_LOCK_CGL_WRITE)

```
> +
> +#define CFT_LOCK_FOR_READ (CFT_LOCK_ATTACH_READ |  \
> +      CFT_LOCK_RMDIR_READ |  \
> +      CFT_LOCK_HIERARCHY_READ |  \
> +      CFT_LOCK_CGL_READ)
> +
> +#define CFT_LOCK_FOR_WRITE (CFT_LOCK_ATTACH_WRITE | \
> +       CFT_LOCK_RMDIR_WRITE | \
> +       CFT_LOCK_HIERARCHY_WRITE | \
> +       CFT_LOCK_CGL_WRITE)
> +
>  struct cftype {
>   /* By convention, the name should begin with the name of the
>    * subsystem, followed by a period */
>   char name[MAX_CFTYPE_NAME];
>   int private;
> +
> + /*
> +  * Determine what locks (if any) are held across calls to
> +  * read_X/write_X callback. See lockmode definitions above
> +  */
> + int lockmode;
> +
>   int (*open) (struct inode *inode, struct file *file);
>   ssize_t (*read) (struct cgroup *cgrp, struct cftype *cft,
>      struct file *file,
> Index: cgroup-2.6.25-mm1/kernel/cgroup.c
> ===================================================================
> --- cgroup-2.6.25-mm1.orig/kernel/cgroup.c
> +++ cgroup-2.6.25-mm1/kernel/cgroup.c
> @@ -1327,38 +1327,65 @@ enum cgroup_filetype {
>   FILE_RELEASE_AGENT,
>  };
>
> -static ssize_t cgroup_write_X64(struct cgroup *cgrp, struct cftype *cft,
> -    struct file *file,
> -    const char __user *userbuf,
> -    size_t nbytes, loff_t *unused_ppos)
> +
> +
> +/**
> + * cgroup_file_lock(). Helper for cgroup read/write methods.
> + * @cgrp:  the cgroup being acted on
> + * @cft:   the control file being written to or read from
> + * *write: true if the access is a write access.
> + *
> + * Takes any necessary locks as requested by the control file's
> + * 'lockmode' field; checks (after locking if necessary) that the
```

```
> + * control group is not in the process of being destroyed.
> + *
> + * Currently all the locking options are implemented in the same way,
> + * by taking cgroup_mutex. Future patches will add finer-grained
> + * locking.
> + *
> + * Calls to cgroup_file_lock() should always be paired with calls to
> + * cgroup_file_unlock(), even if cgroup_file_lock() returns an error.
> + */
> +
> +static int cgroup_file_lock(struct cgroup *cgrp, struct cftype *cft, int write)
>  {
> - char buffer[64];
> - int retval = 0;
> - char *end;
> + int mask = write ? CFT_LOCK_FOR_WRITE : CFT_LOCK_FOR_READ;
> + BUILD_BUG_ON(CFT_LOCK_FOR_READ != (CFT_LOCK_FOR_WRITE >> 1));
>
> - if (!nbytes)
> -  return -EINVAL;
> - if (nbytes >= sizeof(buffer))
> -  return -E2BIG;
> - if (copy_from_user(buffer, userbuf, nbytes))
> -  return -EFAULT;
> + if (cft->lockmode & mask)
> +  mutex_lock(&cgroup_mutex);
> + if (cgroup_is_removed(cgrp))
> +  return -ENODEV;
> + return 0;
> +}
> +
> +/**
> + * cgroup_file_unlock(): undoes the effect of cgroup_file_lock()
> + */
> +
> +static void cgroup_file_unlock(struct cgroup *cgrp, struct cftype *cft,
> +         int write)
> +{
> + int mask = write ? CFT_LOCK_FOR_WRITE : CFT_LOCK_FOR_READ;
> + if (cft->lockmode & mask)
> +  mutex_unlock(&cgroup_mutex);
> +}
>
> - buffer[nbytes] = 0;     /* nul-terminate */
> - strstrip(buffer);
> +static ssize_t cgroup_write_X64(struct cgroup *cgrp, struct cftype *cft,
> +    const char *buffer)
> +{
```

```
> + char *end;
>   if (cft->write_u64) {
>    u64 val = simple_strtoull(buffer, &end, 0);
>    if (*end)
>     return -EINVAL;
> -  retval = cft->write_u64(cgrp, cft, val);
> +  return cft->write_u64(cgrp, cft, val);
>   } else {
>    s64 val = simple_strtoll(buffer, &end, 0);
>    if (*end)
>     return -EINVAL;
> -  retval = cft->write_s64(cgrp, cft, val);
> +  return cft->write_s64(cgrp, cft, val);
>   }
> - if (!retval)
> -  retval = nbytes;
> - return retval;
>  }
>
>  static ssize_t cgroup_common_file_write(struct cgroup *cgrp,
> @@ -1426,47 +1453,82 @@ out1:
>   return retval;
>  }
>
> -static ssize_t cgroup_file_write(struct file *file, const char __user *buf,
> +static ssize_t cgroup_file_write(struct file *file, const char __user *userbuf,
>        size_t nbytes, loff_t *ppos)
>  {
>   struct cftype *cft = __d_cft(file->f_dentry);
>   struct cgroup *cgrp = __d_cgrp(file->f_dentry->d_parent);
> -
> - if (!cft || cgroup_is_removed(cgrp))
> -  return -ENODEV;
> - if (cft->write)
> -  return cft->write(cgrp, cft, file, buf, nbytes, ppos);
> - if (cft->write_u64 || cft->write_s64)
> -  return cgroup_write_X64(cgrp, cft, file, buf, nbytes, ppos);
> - if (cft->trigger) {
> -  int ret = cft->trigger(cgrp, (unsigned int)cft->private);
> -  return ret ? ret : nbytes;
> + ssize_t retval;
> + char static_buffer[64];
> + char *buffer = static_buffer;
> + ssize_t max_bytes = sizeof(static_buffer) - 1;
> + if (!cft->write && !cft->trigger) {
> +  if (!nbytes)
> +   return -EINVAL;
> +  if (nbytes >= max_bytes)
```

> +   return -E2BIG;
> + if (nbytes >= sizeof(static_buffer)) {

afaict this can't happen - we would have already returned -E2BIG?

> +   /* +1 for nul-terminator */
> +   buffer = kmalloc(nbytes + 1, GFP_KERNEL);
> +   if (buffer == NULL)
> +   return -ENOMEM;
> + }
> + if (copy_from_user(buffer, userbuf, nbytes)) {
> +   retval = -EFAULT;
> +   goto out_free;
> + }
> + buffer[nbytes] = 0; /* nul-terminate */
> + strstrip(buffer); /* strip -just- trailing whitespace */
>   }
> - return -EINVAL;
> -}

I'm trying to work out what protects static_buffer?

Why does it need to be static anyway? 64 bytes on-stack is OK.

> -static ssize_t cgroup_read_u64(struct cgroup *cgrp, struct cftype *cft,
> -   struct file *file,
> -   char __user *buf, size_t nbytes,
> -   loff_t *ppos)
> -{
> - char tmp[64];
> - u64 val = cft->read_u64(cgrp, cft);
> - int len = sprintf(tmp, "%llu\n", (unsigned long long) val);
> + retval = cgroup_file_lock(cgrp, cft, 1);
> + if (retval)
> + goto out_unlock;
>
> - return simple_read_from_buffer(buf, nbytes, ppos, tmp, len);
> + if (cft->write)
> + retval = cft->write(cgrp, cft, file, userbuf, nbytes, ppos);
> + else if (cft->write_u64 || cft->write_s64)
> + retval = cgroup_write_X64(cgrp, cft, buffer);
> + else if (cft->trigger)
> + retval = cft->trigger(cgrp, (unsigned int)cft->private);
> + else
> + retval = -EINVAL;
> + if (retval == 0)
> + retval = nbytes;
> + out_unlock:

> + cgroup_file_unlock(cgrp, cft, 1);
> + out_free:
> + if (buffer != static_buffer)
> + kfree(buffer);
> + return retval;
> }

_____
Containers mailing list
Containers@lists.linux-foundation.org
https://lists.linux-foundation.org/mailman/listinfo/containers

---

Subject: Re: [RFC/PATCH 2/8]: CGroup Files: Add a cgroup write_string control file method
Posted by akpm on Tue, 13 May 2008 20:07:10 GMT
View Forum Message <> Reply to Message

On Mon, 12 May 2008 23:37:09 -0700
menage@google.com wrote:

> This patch adds a write_string() method for cgroups control files. The
> semantics are that a buffer is copied from userspace to kernelspace
> and the handler function invoked on that buffer. Any control group
> locking is done after the copy from userspace has occurred. The buffer
> is guaranteed to be nul-terminated, and no longer than max_write_len
> (defaulting to 64 bytes if unspecified). Later patches will convert
> existing raw file write handlers in control group subsystems to use
> this method.
>

nits:

>
> ---
> include/linux/cgroup.h | 10 ++++++++++
> kernel/cgroup.c | 5 ++++-
> 2 files changed, 14 insertions(+), 1 deletion(-)
>
> Index: cgroup-2.6.25-mm1/include/linux/cgroup.h
> ===
> --- cgroup-2.6.25-mm1.orig/include/linux/cgroup.h
> +++ cgroup-2.6.25-mm1/include/linux/cgroup.h
> @@ -281,6 +281,10 @@ struct cftype {
>   */
>   int lockmode;
>
> + /* If non-zero, defines the maximum length of string that can

> + * be passed to write_string; defaults to 64 */
> + int max_write_len;

would size_t be a more appropriate type?

>   int (*open) (struct inode *inode, struct file *file);
>   ssize_t (*read) (struct cgroup *cgrp, struct cftype *cft,
>   struct file *file,
> @@ -323,6 +327,12 @@ struct cftype {
>   * write_s64() is a signed version of write_u64()
>   */
>   int (*write_s64) (struct cgroup *cgrp, struct cftype *cft, s64 val);

s/) (/)(/ would be more conventional.

> + /*
> + * write_string() is passed a nul-terminated kernelspace
> + * buffer of maximum length determined by max_write_len
> + */
> + int (*write_string) (struct cgroup *cgrp, struct cftype *cft,
> +   char *buffer);

Should these return size_t?

>   /*
>   * trigger() callback can be used to get some kick from the
> Index: cgroup-2.6.25-mm1/kernel/cgroup.c
> ===
> --- cgroup-2.6.25-mm1.orig/kernel/cgroup.c
> +++ cgroup-2.6.25-mm1/kernel/cgroup.c
> @@ -1461,7 +1461,7 @@ static ssize_t cgroup_file_write(struct
>   ssize_t retval;
>   char static_buffer[64];
>   char *buffer = static_buffer;
> - ssize_t max_bytes = sizeof(static_buffer) - 1;
> + ssize_t max_bytes = cft->max_write_len ?: sizeof(static_buffer) - 1;

A blank line between end-of-locals and start-of-code is conventional
and, IMO, easier on the eye.

Does gcc actually generate better code with that x?:y thing? Because
it always makes me pause and scratch my head.

>   if (!cft->write && !cft->trigger) {
>   if (!nbytes)
>   return -EINVAL;
> @@ -1489,6 +1489,8 @@ static ssize_t cgroup_file_write(struct
>   retval = cft->write(cgrp, cft, file, userbuf, nbytes, ppos);

> else if (cft->write_u64 || cft->write_s64)
>   retval = cgroup_write_X64(cgrp, cft, buffer);
> + else if (cft->write_string)
> + retval = cft->write_string(cgrp, cft, buffer);
> else if (cft->trigger)
>   retval = cft->trigger(cgrp, (unsigned int)cft->private);
> else
> @@ -1651,6 +1653,7 @@ static struct file_operations cgroup_seq
> .read = seq_read,
> .llseek = seq_lseek,
> .release = cgroup_seqfile_release,
> + .write = cgroup_file_write,
> };
>
> static int cgroup_file_open(struct inode *inode, struct file *file)

_____
Containers mailing list
Containers@lists.linux-foundation.org
https://lists.linux-foundation.org/mailman/listinfo/containers

---

Subject: Re: [RFC/PATCH 3/8]: CGroup Files: Move the release_agent file to use typed handlers
Posted by akpm on Tue, 13 May 2008 20:08:33 GMT
View Forum Message <> Reply to Message

On Mon, 12 May 2008 23:37:10 -0700
menage@google.com wrote:

> + agentbuf = kmalloc(PATH_MAX, GFP_KERNEL);
> + if (!agentbuf)
> +   goto continue_free;
> + strcpy(agentbuf, cgrp->root->release_agent_path);

Did we need to allocate all that memory, or would kstrdup() suffice?

_____
Containers mailing list
Containers@lists.linux-foundation.org
https://lists.linux-foundation.org/mailman/listinfo/containers

---

Subject: Re: [RFC/PATCH 6/8]: CGroup Files: Remove cpuset_common_file_write()
Posted by akpm on Tue, 13 May 2008 20:11:34 GMT
View Forum Message <> Reply to Message

On Mon, 12 May 2008 23:37:13 -0700

menage@google.com wrote:

> @@ -1412,14 +1355,18 @@ static struct cftype files[] = {
> {
> .name = "cpus",
> .read = cpuset_common_file_read,
> - .write = cpuset_common_file_write,
> + .write_string = update_cpumask,
> + .max_write_len = (100U + 6 * NR_CPUS),

hm, magic handwavy surely-enough-for-anyone constants.

What's going on here?

> + .lockmode = CFT_LOCK_CGL_WRITE,
> .private = FILE_CPULIST,
> },

_____
Containers mailing list
Containers@lists.linux-foundation.org
https://lists.linux-foundation.org/mailman/listinfo/containers

---

Subject: Re: [RFC/PATCH 1/8]: CGroup Files: Add locking mode to cgroups control files
Posted by Matt Helsley on Tue, 13 May 2008 20:38:58 GMT
View Forum Message <> Reply to Message

On Tue, 2008-05-13 at 13:01 -0700, Andrew Morton wrote:
> Fear, doubt and resistance!
>
> On Mon, 12 May 2008 23:37:08 -0700
> menage@google.com wrote:
>
> > Different cgroup files have different stability requirements of the
> > cgroups framework while the handler is running; currently most
> > subsystems that don't have their own internal synchronization just
> > call cgroup_lock()/cgroup_unlock(), which takes the global cgroup_mutex.
> >
> > This patch introduces a range of locking modes that can be requested
> > by a control file; currently these are all implemented internally by
> > taking cgroup_mutex, but expressing the intention will make it simpler
> > to move to a finer-grained locking scheme in the future.
> >
>
> This, umm, doesn't seem to do much to make the kernel a simpler place.
>
> Do we expect to gain much from this? Hope so... What?

>
> > Index: cgroup-2.6.25-mm1/include/linux/cgroup.h
> > ===
> > --- cgroup-2.6.25-mm1.orig/include/linux/cgroup.h
> > +++ cgroup-2.6.25-mm1/include/linux/cgroup.h
> > @@ -200,11 +200,87 @@ struct cgroup_map_cb {
> >   */
> >
> > #define MAX_CFTYPE_NAME 64
> > +
> > +/* locking modes for control files.
> > + *
> > + * These determine what level of guarantee the file handler wishes
> > + * cgroups to provide about the stability of control group entities
> > + * for the duration of the handler callback.
> > + *
> > + * The minimum guarantee is that the subsystem state for this
> > + * subsystem will not be freed (via a call to the subsystem's
> > + * destroy() callback) until after the control file handler
> > + * returns. This guarantee is provided by the fact that the open
> > + * dentry for the control file keeps its parent (cgroup) dentry alive,
> > + * which in turn keeps the cgroup object from being actually freed
> > + * (although it can be moved into the removed state in the
> > + * meantime). This is suitable for subsystems that completely control
> > + * their own synchronization.
> > + *
> > + * Other possible guarantees are given below.
> > + *
> > + * XXX_READ bits are used for a read operation on the control file,
> > + * XXX_WRITE bits are used for a write operation on the control file
> > + */
>
> Vague handwaving: lockdep doesn't know anything about any of this.
> Whereas if we were more conventional in using separate locks and
> suitable lock types for each application, we would retain full lockdep
> coverage.
>
> > +/*
> > + * CFT_LOCK_ATTACH_(READ|WRITE): This operation will not run
> > + * concurrently with a task movement into or out of this cgroup.
> > + */
> > +#define CFT_LOCK_ATTACH_READ 1
> > +#define CFT_LOCK_ATTACH_WRITE 2
> > +#define CFT_LOCK_ATTACH (CFT_LOCK_ATTACH_READ |
CFT_LOCK_ATTACH_WRITE)
> > +
> > +/*
> > + * CFT_LOCK_RMDIR_(READ|WRITE): This operation will not run

> > + * concurrently with the removal of the affected cgroup.
> > + */
> > +#define CFT_LOCK_RMDIR_READ 4
> > +#define CFT_LOCK_RMDIR_WRITE 8
> > +#define CFT_LOCK_RMDIR (CFT_LOCK_RMDIR_READ | CFT_LOCK_RMDIR_WRITE)
> > +
> > +/*
> > + * CFT_LOCK_HIERARCHY_(READ|WRITE): This operation will not run
> > + * concurrently with a cgroup creation or removal in this hierarchy,
> > + * or a bind/move/unbind for this subsystem.
> > + */
> > +#define CFT_LOCK_HIERARCHY_READ 16
> > +#define CFT_LOCK_HIERARCHY_WRITE 32
> > +#define CFT_LOCK_HIERARCHY (CFT_LOCK_HIERARCHY_READ |
CFT_LOCK_HIERARCHY_WRITE)
> > +
> > +/*
> > + * CFT_LOCK_CGL_(READ|WRITE): This operation is called with
> > + * cgroup_lock() held; it will not run concurrently with any of the
> > + * above operations in any cgroup/hierarchy. This should be considered
> > + * to be the BKL of cgroups - it should be avoided if you can use
> > + * finer-grained locking
> > + */
> > +#define CFT_LOCK_CGL_READ 64
> > +#define CFT_LOCK_CGL_WRITE 128
> > +#define CFT_LOCK_CGL (CFT_LOCK_CGL_READ | CFT_LOCK_CGL_WRITE)
> > +
> > +#define CFT_LOCK_FOR_READ (CFT_LOCK_ATTACH_READ | \
> > +   CFT_LOCK_RMDIR_READ | \
> > +   CFT_LOCK_HIERARCHY_READ | \
> > +   CFT_LOCK_CGL_READ)
> > +
> > +#define CFT_LOCK_FOR_WRITE (CFT_LOCK_ATTACH_WRITE | \
> > +   CFT_LOCK_RMDIR_WRITE | \
> > +   CFT_LOCK_HIERARCHY_WRITE | \
> > +   CFT_LOCK_CGL_WRITE)
> > +
> > struct cftype {
> >   /* By convention, the name should begin with the name of the
> >   * subsystem, followed by a period */
> >   char name[MAX_CFTYPE_NAME];
> >   int private;
> > +
> > + /*
> > + * Determine what locks (if any) are held across calls to
> > + * read_X/write_X callback. See lockmode definitions above
> > + */
> > + int lockmode;

> > +
> > int (*open) (struct inode *inode, struct file *file);
> > ssize_t (*read) (struct cgroup *cgrp, struct cftype *cft,
> > struct file *file,
> > Index: cgroup-2.6.25-mm1/kernel/cgroup.c
> > ===
> > --- cgroup-2.6.25-mm1.orig/kernel/cgroup.c
> > +++ cgroup-2.6.25-mm1/kernel/cgroup.c
> > @@ -1327,38 +1327,65 @@ enum cgroup_filetype {
> > FILE_RELEASE_AGENT,
> > };
> >
> > -static ssize_t cgroup_write_X64(struct cgroup *cgrp, struct cftype *cft,
> > - struct file *file,
> > - const char __user *userbuf,
> > - size_t nbytes, loff_t *unused_ppos)
> > +
> > +
> > +/**
> > + * cgroup_file_lock(). Helper for cgroup read/write methods.
> > + * @cgrp: the cgroup being acted on
> > + * @cft: the control file being written to or read from
> > + * *write: true if the access is a write access.
> > + *
> > + * Takes any necessary locks as requested by the control file's
> > + * 'lockmode' field; checks (after locking if necessary) that the
> > + * control group is not in the process of being destroyed.
> > + *
> > + * Currently all the locking options are implemented in the same way,
> > + * by taking cgroup_mutex. Future patches will add finer-grained
> > + * locking.
> > + *
> > + * Calls to cgroup_file_lock() should always be paired with calls to
> > + * cgroup_file_unlock(), even if cgroup_file_lock() returns an error.
> > + */
> > +
> > +static int cgroup_file_lock(struct cgroup *cgrp, struct cftype *cft, int write)
> > {
> > - char buffer[64];
> > - int retval = 0;
> > - char *end;
> > + int mask = write ? CFT_LOCK_FOR_WRITE : CFT_LOCK_FOR_READ;
> > + BUILD_BUG_ON(CFT_LOCK_FOR_READ != (CFT_LOCK_FOR_WRITE >> 1));
> >
> > - if (!nbytes)
> > - return -EINVAL;
> > - if (nbytes >= sizeof(buffer))
> > - return -E2BIG;

```
> > - if (copy_from_user(buffer, userbuf, nbytes))
> > -  return -EFAULT;
> > + if (cft->lockmode & mask)
> > +  mutex_lock(&cgroup_mutex);
> > + if (cgroup_is_removed(cgrp))
> > +  return -ENODEV;
> > + return 0;
> > +}
> > +
> > +/**
> > + * cgroup_file_unlock(): undoes the effect of cgroup_file_lock()
> > + */
> > +
> > +static void cgroup_file_unlock(struct cgroup *cgrp, struct cftype *cft,
> > +         int write)
> > +{
> > + int mask = write ? CFT_LOCK_FOR_WRITE : CFT_LOCK_FOR_READ;
> > + if (cft->lockmode & mask)
> > +  mutex_unlock(&cgroup_mutex);
> > +}
> >
> > - buffer[nbytes] = 0;     /* nul-terminate */
> > - strstrip(buffer);
> > +static ssize_t cgroup_write_X64(struct cgroup *cgrp, struct cftype *cft,
> > +    const char *buffer)
> > +{
> > + char *end;
> >   if (cft->write_u64) {
> >    u64 val = simple_strtoull(buffer, &end, 0);
> >    if (*end)
> >     return -EINVAL;
> > -  retval = cft->write_u64(cgrp, cft, val);
> > +  return cft->write_u64(cgrp, cft, val);
> >   } else {
> >    s64 val = simple_strtoll(buffer, &end, 0);
> >    if (*end)
> >     return -EINVAL;
> > -  retval = cft->write_s64(cgrp, cft, val);
> > +  return cft->write_s64(cgrp, cft, val);
> >   }
> > - if (!retval)
> > -  retval = nbytes;
> > - return retval;
> >  }
> >
> >  static ssize_t cgroup_common_file_write(struct cgroup *cgrp,
> > @@ -1426,47 +1453,82 @@ out1:
> >   return retval;
```

```
> >  }
> >
> > -static ssize_t cgroup_file_write(struct file *file, const char __user *buf,
> > +static ssize_t cgroup_file_write(struct file *file, const char __user *userbuf,
> >        size_t nbytes, loff_t *ppos)
> >  {
> >   struct cftype *cft = __d_cft(file->f_dentry);
> >   struct cgroup *cgrp = __d_cgrp(file->f_dentry->d_parent);
> > -
> > - if (!cft || cgroup_is_removed(cgrp))
> > -  return -ENODEV;
> > - if (cft->write)
> > -  return cft->write(cgrp, cft, file, buf, nbytes, ppos);
> > - if (cft->write_u64 || cft->write_s64)
> > -  return cgroup_write_X64(cgrp, cft, file, buf, nbytes, ppos);
> > - if (cft->trigger) {
> > -  int ret = cft->trigger(cgrp, (unsigned int)cft->private);
> > -  return ret ? ret : nbytes;
> > + ssize_t retval;
> > + char static_buffer[64];
> > + char *buffer = static_buffer;
> > + ssize_t max_bytes = sizeof(static_buffer) - 1;
> > + if (!cft->write && !cft->trigger) {
> > +  if (!nbytes)
> > +   return -EINVAL;
> > +  if (nbytes >= max_bytes)
> > +   return -E2BIG;
> > +  if (nbytes >= sizeof(static_buffer)) {
>
> afaict this can't happen - we would have already returned -E2BIG?
>
> > +   /* +1 for nul-terminator */
> > +   buffer = kmalloc(nbytes + 1, GFP_KERNEL);
> > +   if (buffer == NULL)
> > +    return -ENOMEM;
> > +  }
> > +  if (copy_from_user(buffer, userbuf, nbytes)) {
> > +   retval = -EFAULT;
> > +   goto out_free;
> > +  }
> > +  buffer[nbytes] = 0; /* nul-terminate */
> > +  strstrip(buffer); /* strip -just- trailing whitespace */
> >   }
> > - return -EINVAL;
> > -}
>
> I'm trying to work out what protects static_buffer?
```

One of us must be having a brain cramp because it looks to me like the
buffer doesn't need protection -- it's on the stack. It's probably me
but I'm just not seeing how this use is unsafe..

> Why does it need to be static anyway? 64 bytes on-stack is OK.

Uh, it is on stack. It doesn't use the C keyword "static". It's just
poorly-named.

<snip>

Cheers,
 -Matt Helsley

_____
Containers mailing list
Containers@lists.linux-foundation.org
https://lists.linux-foundation.org/mailman/listinfo/containers

---

Subject: Re: [RFC/PATCH 1/8]: CGroup Files: Add locking mode to cgroups control files
Posted by akpm on Tue, 13 May 2008 20:43:54 GMT
View Forum Message <> Reply to Message

On Tue, 13 May 2008 13:38:58 -0700
Matthew Helsley <matthltc@us.ibm.com> wrote:

> > > + char static_buffer[64];
>
> > I'm trying to work out what protects static_buffer?
>
> One of us must be having a brain cramp because it looks to me like the
> buffer doesn't need protection -- it's on the stack. It's probably me
> but I'm just not seeing how this use is unsafe..

doh. Well it had me going...

> Uh, it is on stack. It doesn't use the C keyword "static". It's just
> poorly-named.

That depends upon one's objectives in choosing a name ;)

_____
Containers mailing list
Containers@lists.linux-foundation.org
https://lists.linux-foundation.org/mailman/listinfo/containers

---

Subject: Re: [RFC/PATCH 2/8]: CGroup Files: Add a cgroup write_string control file method
Posted by Matt Helsley on Tue, 13 May 2008 20:44:11 GMT
View Forum Message <> Reply to Message

On Mon, 2008-05-12 at 23:37 -0700, menage@google.com wrote:
> plain text document attachment (cgroup_write_string.patch)
> This patch adds a write_string() method for cgroups control files. The
> semantics are that a buffer is copied from userspace to kernelspace
> and the handler function invoked on that buffer. Any control group
> locking is done after the copy from userspace has occurred. The buffer
> is guaranteed to be nul-terminated, and no longer than max_write_len
> (defaulting to 64 bytes if unspecified). Later patches will convert
> existing raw file write handlers in control group subsystems to use
> this method.
>
> Signed-off-by: Paul Menage <menage@google.com>

I haven't looked at this very thoroughly but I think the goal of this
patch is good. Can't this patch and the conversions precede your locking
changes?

Cheers,
 -Matt Helsley

_____
Containers mailing list
Containers@lists.linux-foundation.org
https://lists.linux-foundation.org/mailman/listinfo/containers

Subject: Re: [RFC/PATCH 2/8]: CGroup Files: Add a cgroup write_string control file method
Posted by Paul Menage on Tue, 13 May 2008 21:01:01 GMT
View Forum Message <> Reply to Message

On Tue, May 13, 2008 at 1:07 PM, Andrew Morton
<akpm@linux-foundation.org> wrote:
> >
> > +   /* If non-zero, defines the maximum length of string that can
> > +   * be passed to write_string; defaults to 64 */
> > +   int max_write_len;
>
> would size_t be a more appropriate type?
>

Probably overkill, but I guess it's technically more correct. Updated
for the next version of these patches.

>
> s/) (/)(/ would be more conventional.
>

OK, I've updated this and the other extraneous spaces in a separate patch.

>
> > +   /*
> > +   * write_string() is passed a nul-terminated kernelspace
> > +   * buffer of maximum length determined by max_write_len
> > +   */
> > +   int (*write_string) (struct cgroup *cgrp, struct cftype *cft,
> > +   char *buffer);
>
> Should these return size_t?

No, it returns 0 or a -ve error code. I've added a comment to this effect.
> >   char *buffer = static_buffer;
> > -   ssize_t max_bytes = sizeof(static_buffer) - 1;
> > +   ssize_t max_bytes = cft->max_write_len ?: sizeof(static_buffer) - 1;
>
> A blank line between end-of-locals and start-of-code is conventional
> and, IMO, easier on the eye.
>
> Does gcc actually generate better code with that x?:y thing?

I doubt it - but I felt that it made the code a bit clearer since it
reduces repetition. I can change it to

```
 size_t max_bytes =  cft->max_write_len;

 if (!max_bytes)
  max_bytes = sizeof(static_buffer) - 1;
```

Paul

_____
Containers mailing list
Containers@lists.linux-foundation.org
https://lists.linux-foundation.org/mailman/listinfo/containers

---

Subject: Re: [RFC/PATCH 1/8]: CGroup Files: Add locking mode to cgroups control files
Posted by Paul Menage on Tue, 13 May 2008 21:07:40 GMT
View Forum Message <> Reply to Message

On Tue, May 13, 2008 at 2:23 AM, Li Zefan <lizf@cn.fujitsu.com> wrote:

> > +
> > +/**
> > + * cgroup_file_lock(). Helper for cgroup read/write methods.
> > + * @cgrp: the cgroup being acted on
> > + * @cft: the control file being written to or read from
> > + * *write: true if the access is a write access.
>
> s/*write/@write
>

Fixed.

> > @@ -1518,16 +1580,21 @@ static ssize_t cgroup_file_read(struct f
> >   struct cftype *cft = __d_cft(file->f_dentry);
> >   struct cgroup *cgrp = __d_cgrp(file->f_dentry->d_parent);
> >
> > -   if (!cft || cgroup_is_removed(cgrp))
> > +   if (cgroup_is_removed(cgrp))
> >   return -ENODEV;
> >
>
> This check seems redundant now.
>

It's not needed for safety, but it doesn't seem to hurt to check
cgroup_is_removed() prior to doing any copying, since we'll fail after
copying anyway if cgroup_is_removed() returns true (once we've taken
any relevant locks).

Paul

Subject: Re: [RFC/PATCH 1/8]: CGroup Files: Add locking mode to cgroups control files
Posted by Paul Menage on Tue, 13 May 2008 21:17:29 GMT
View Forum Message <> Reply to Message

On Tue, May 13, 2008 at 1:01 PM, Andrew Morton
<akpm@linux-foundation.org> wrote:
>
> This, umm, doesn't seem to do much to make the kernel a simpler place.
>
> Do we expect to gain much from this? Hope so... What?
>

The goal is to prevent cgroup_mutex becoming a BKL for cgroups, to make it easier for subsystems to lock just the bits that they need to remain stable rather than everything.
>
> Vague handwaving: lockdep doesn't know anything about any of this.
> Whereas if we were more conventional in using separate locks and
> suitable lock types for each application, we would retain full lockdep
> coverage.

That's a good point - I'd not thought about lockdep. That's a good argument in favour of not having the locking done in the framework.

Stepping back a bit, the idea is definitely that where appropriate subsystems will use their own fine-grained locking. E.g. the res_counter abstraction does this already with a spinlock in each res_counter, and cpusets has the global callback_mutex that just synchronizes cpuset operations. But there are some cases where they need a bit of help from cgroups, such as when doing operations that require stable hierarchies, task membership of cgroups, etc.

Right now the default behaviour is to call cgroup_lock(), which I'd like to get away from. Having the framework do the locking results in less need for cleanup code in the subsystem handlers themselves, but that's not an unassailable argument for doing it that way.

This is one of those times that I really long for C++ scope-based destructors, to automatically release locks when you exit a scope.

```
> > +     ssize_t max_bytes = sizeof(static_buffer) - 1;
> > +     if (!cft->write && !cft->trigger) {
> > +             if (!nbytes)
> > +                     return -EINVAL;
> > +             if (nbytes >= max_bytes)
> > +                     return -E2BIG;
> > +             if (nbytes >= sizeof(static_buffer)) {
>
> afaict this can't happen - we would have already returned -E2BIG?
```

You're right, with this patch it can't. The code that makes it necessary to allocate a dynamic buffer gets added in the write_string patch, so I'll move it there. (These two were originally one patch and I guess I did a poor job of splitting them).

```
>
>
> > +                     /* +1 for nul-terminator */
> > +                     buffer = kmalloc(nbytes + 1, GFP_KERNEL);
```

```
> > +                 if (buffer == NULL)
> > +                         return -ENOMEM;
> > +         }
> > +         if (copy_from_user(buffer, userbuf, nbytes)) {
> > +                 retval = -EFAULT;
> > +                 goto out_free;
> > +         }
> > +         buffer[nbytes] = 0;     /* nul-terminate */
> > +         strstrip(buffer);       /* strip -just- trailing whitespace */
> >      }
> > -    return -EINVAL;
> > -}
>
>  I'm trying to work out what protects static_buffer?
>
>  Why does it need to be static anyway?  64 bytes on-stack is OK.
>
```

As Matt observed, this is just a poorly-named variable. How about "small_buffer"?

Paul

Subject: Re: [RFC/PATCH 6/8]: CGroup Files: Remove cpuset_common_file_write()
Posted by Paul Menage on Tue, 13 May 2008 21:27:24 GMT
View Forum Message <> Reply to Message

On Tue, May 13, 2008 at 1:11 PM, Andrew Morton
<akpm@linux-foundation.org> wrote:
> On Mon, 12 May 2008 23:37:13 -0700
> menage@google.com wrote:
>
> > @@ -1412,14 +1355,18 @@ static struct cftype files[] = {
> >   {
> >   .name = "cpus",
> >   .read = cpuset_common_file_read,
> > -   .write = cpuset_common_file_write,
> > +   .write_string = update_cpumask,
> > +   .max_write_len = (100U + 6 * NR_CPUS),
>
> hm, magic handwavy surely-enough-for-anyone constants.

This is just a movement of the check previously in cpuset_common_file_write():

```
 /* Crude upper limit on largest legitimate cpulist user might write. */
 if (nbytes > 100U + 6 * max(NR_CPUS, MAX_NUMNODES))
  return -E2BIG;
```

I'll add more comments. But I think it's a reasonable limit - assuming that you wastefully enumerate each cpu individually, separated by commas, 6 * NR_CPUS is enough for 100K CPUs.

Paul

_____

Containers mailing list
Containers@lists.linux-foundation.org
https://lists.linux-foundation.org/mailman/listinfo/containers

---

Subject: Re: [RFC/PATCH 1/8]: CGroup Files: Add locking mode to cgroups control files

Posted by akpm on Tue, 13 May 2008 21:32:06 GMT

View Forum Message <> Reply to Message

On Tue, 13 May 2008 14:17:29 -0700
"Paul Menage" <menage@google.com> wrote:

> On Tue, May 13, 2008 at 1:01 PM, Andrew Morton
> <akpm@linux-foundation.org> wrote:
> >
> > This, umm, doesn't seem to do much to make the kernel a simpler place.
> >
> > Do we expect to gain much from this? Hope so... What?
> >
>
> The goal is to prevent cgroup_mutex becoming a BKL for cgroups, to
> make it easier for subsystems to lock just the bits that they need to
> remain stable rather than everything.

OK.

But do we ever expect that cgroup_mutex will be taken with much frequency, or held for much time? If it's only taken during, say, configuration of a group or during a query of that configuration then perhaps we'll be OK.

otoh a per-cgroup lock would seem more appropriate than a global.

> >
> Vague handwaving: lockdep doesn't know anything about any of this.
> Whereas if we were more conventional in using separate locks and

> > suitable lock types for each application, we would retain full lockdep
> > coverage.
>
> That's a good point - I'd not thought about lockdep. That's a good
> argument in favour of not having the locking done in the framework.
>
> Stepping back a bit, the idea is definitely that where appropriate
> subsystems will use their own fine-grained locking. E.g. the
> res_counter abstraction does this already with a spinlock in each
> res_counter, and cpusets has the global callback_mutex that just
> synchronizes cpuset operations. But there are some cases where they
> need a bit of help from cgroups, such as when doing operations that
> require stable hierarchies, task membership of cgroups, etc.
>
> Right now the default behaviour is to call cgroup_lock(), which I'd
> like to get away from. Having the framework do the locking results in
> less need for cleanup code in the subsystem handlers themselves, but
> that's not an unassailable argument for doing it that way.

Yes, caller-provided locking is the usual pattern in-kernel. Based on
painful experience :(

> > I'm trying to work out what protects static_buffer?
> >
> > Why does it need to be static anyway? 64 bytes on-stack is OK.
> >
>
> As Matt observed, this is just a poorly-named variable. How about
> "small_buffer"?

local_buffer ;)

Subject: Re: [RFC/PATCH 3/8]: CGroup Files: Move the release_agent file to use typed handlers

Posted by Paul Menage on Tue, 13 May 2008 21:32:57 GMT

View Forum Message <> Reply to Message

On Tue, May 13, 2008 at 1:08 PM, Andrew Morton
<akpm@linux-foundation.org> wrote:
> On Mon, 12 May 2008 23:37:10 -0700
> menage@google.com wrote:
>
> > > + agentbuf = kmalloc(PATH_MAX, GFP_KERNEL);

> > + if (!agentbuf)
> > + goto continue_free;
> > + strcpy(agentbuf, cgrp->root->release_agent_path);
>
> Did we need to allocate all that memory, or would kstrdup() suffice?
>

kstrdup() would be fine. Fixed.

Paul

_____

Containers mailing list
Containers@lists.linux-foundation.org
https://lists.linux-foundation.org/mailman/listinfo/containers

---

Subject: Re: [RFC/PATCH 1/8]: CGroup Files: Add locking mode to cgroups control files
Posted by Paul Menage on Tue, 13 May 2008 21:46:05 GMT
View Forum Message <> Reply to Message

On Tue, May 13, 2008 at 2:32 PM, Andrew Morton
<akpm@linux-foundation.org> wrote:
>
> But do we ever expect that cgroup_mutex will be taken with much
> frequency, or held for much time? If it's only taken during, say,
> configuration of a group or during a query of that configuration then
> perhaps we'll be OK.

I'm not so worried about contention on the userspace configuration side - more the case of configuration operations stalling subsystem-initiated operations. An example of that would be the case of hierarchical reclaim in the memory controller, where it needs to be able to traverse up and down the hierarchy without the hierarchy changing under its feet.

I'll dump the implicit lock_mode support and make it more explicit in the subsystem handlers.

Paul

_____

Containers mailing list
Containers@lists.linux-foundation.org
https://lists.linux-foundation.org/mailman/listinfo/containers

---

Subject: Re: [RFC/PATCH 1/8]: CGroup Files: Add locking mode to cgroups control

files
Posted by Li Zefan on Wed, 14 May 2008 01:30:40 GMT
View Forum Message <> Reply to Message

>> > @@ -1518,16 +1580,21 @@ static ssize_t cgroup_file_read(struct f
>> > struct cftype *cft = __d_cft(file->f_dentry);
>> > struct cgroup *cgrp = __d_cgrp(file->f_dentry->d_parent);
>> >
>> > - if (!cft || cgroup_is_removed(cgrp))
>> > + if (cgroup_is_removed(cgrp))
>> > return -ENODEV;
>> >
>>
>> This check seems redundant now.
>>
>
> It's not needed for safety, but it doesn't seem to hurt to check
> cgroup_is_removed() prior to doing any copying, since we'll fail after
> copying anyway if cgroup_is_removed() returns true (once we've taken
> any relevant locks).
>

It's a bit odd to me that cgroup_is_removed() is removed in cgroup_file_write(),
but it's ressered here.

_____
Containers mailing list
Containers@lists.linux-foundation.org
https://lists.linux-foundation.org/mailman/listinfo/containers

Subject: Re: [RFC/PATCH 1/8]: CGroup Files: Add locking mode to cgroups control files
Posted by Paul Menage on Wed, 14 May 2008 01:40:12 GMT
View Forum Message <> Reply to Message

On Tue, May 13, 2008 at 6:30 PM, Li Zefan <lizf@cn.fujitsu.com> wrote:
>
> It's a bit odd to me that cgroup_is_removed() is removed in cgroup_file_write(),
> but it's ressered here.
>

Fair point. It's probably better for me to be consistent.

Paul

_____
Containers mailing list
Containers@lists.linux-foundation.org

Subject: Re: [RFC/PATCH 1/8]: CGroup Files: Add locking mode to cgroups control files
Posted by Paul Jackson on Wed, 14 May 2008 01:59:52 GMT
View Forum Message <> Reply to Message

Andrew wrote:
> As Matt observed, this is just a poorly-named variable.

How about the following code for cgroup_file_write():

```
 char buf[64];  /* avoid kmalloc() in small cases */
 char *p;  /* buf[] or kmalloc'd buffer */

 ...

 if (nbytes < sizeof(buf)) {
  p = buf;
 } else {
  p = kmalloc(nbytes + 1, GFP_KERNEL);
  if (p == NULL)
   return -ENOMEM;
 }
```

Possible advantages of above code:

 * Uses short names for local variables of limited scope.

 * Doesn't set p until needed, so as:
1) to highlight the symmetry of its setting, to either buf[]
   (small cases) or to a kmalloc'd buffer (large cases), and
2) to avoid implying that p needs to be set in the intervening
   "..." code above.

 * Comments variable declarations.

--
                I won't rest till it's the best ...
                Programmer, Linux Scalability
                Paul Jackson <pj@sgi.com> 1.940.382.4214