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