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