Subject: [PATCH 3/4] Container Freezer: Implement freezer cgroup subsystem
Posted by Matt Helsley on Mon, 07 Jul 2008 22:58:26 GMT
View Forum Message <> Reply to Message

From: Cedric Le Goater <clg@fr.ibm.com>
Subject: [PATCH 3/4] Container Freezer: Implement freezer cgroup subsystem

This patch implements a new freezer subsystem in the control groups framework.
It provides a way to stop and resume execution of all tasks in a cgroup by
writing in the cgroup filesystem.

This is the basic mechanism which should do the right thing for user space task
in a simple scenario.

It's important to note that freezing can be incomplete. In that case we return
EBUSY. This means that some tasks in the cgroup are busy doing something that
prevents us from completely freezing the cgroup at this time. After EBUSY,
the cgroup will remain partially frozen -- reflected by freezer.state reporting
"FREEZING" when read. The state will remain "FREEZING" until one of these
things happens:

 1) Userspace cancels the freezing operation by writing "RUNNING" to
  the freezer.state file
 2) Userspace retries the freezing operation by writing "FROZEN" to
  the freezer.state file (writing "FREEZING" is not legal
  and returns EIO)
 3) The tasks that blocked the cgroup from entering the "FROZEN"
  state disappear from the cgroup's set of tasks.

Signed-off-by: Cedric Le Goater <clg@fr.ibm.com>
Signed-off-by: Matt Helsley <matthltc@us.ibm.com>
Tested-by: Matt Helsley <matthltc@us.ibm.com>

Changelog:
v2:
 Moved the "kill" file into a separate cgroup subsystem (signal) and
  it's own patch.
 Changed the name of the file from freezer.freeze to freezer.state.
 Switched from taking 1 and 0 as input to the strings "FROZEN" and
  "RUNNING", respectively. This helps keep the interface
  human-usable if/when we need to more states.
 Checked that stopped or interrupted is "frozen enough"
  Since try_to_freeze() is called upon wakeup of these tasks
  this should be fine. This idea comes from recent changes to
  the freezer.
 Checked that if (task == current) whilst freezing cgroup we're ok
 Fixed bug where -EBUSY would always be returned when freezing
 Added code to handle userspace retries for any remaining -EBUSY

```
 include/linux/cgroup_freezer.h |   71 +++++++++
 include/linux/cgroup_subsys.h  |    6
 include/linux/freezer.h        |   16 +-
 init/Kconfig                   |    7
 kernel/Makefile                |    1
 kernel/cgroup_freezer.c        |  317 ++++++++++++++++++++++++++++++++++++++++++
 6 files changed, 414 insertions(+), 4 deletions(-)

Index: linux-2.6.26-rc5-mm2/include/linux/cgroup_freezer.h
===================================================================
--- /dev/null
+++ linux-2.6.26-rc5-mm2/include/linux/cgroup_freezer.h
@@ -0,0 +1,71 @@
+#ifndef _LINUX_CGROUP_FREEZER_H
+#define _LINUX_CGROUP_FREEZER_H
+/*
+ * cgroup_freezer.h -  control group freezer subsystem interface
+ *
+ * Copyright IBM Corporation, 2007
+ *
+ * Author : Cedric Le Goater <clg@fr.ibm.com>
+ *
+ * This program is free software; you can redistribute it and/or modify it
+ * under the terms of version 2.1 of the GNU Lesser General Public License
+ * as published by the Free Software Foundation.
+ *
+ * This program is distributed in the hope that it would be useful, but
+ * WITHOUT ANY WARRANTY; without even the implied warranty of
+ * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
+ */
+
+#include <linux/cgroup.h>
+
+#ifdef CONFIG_CGROUP_FREEZER
+
+enum freezer_state {
+ STATE_RUNNING = 0,
+ STATE_FREEZING,
+ STATE_FROZEN,
+};
+
+struct freezer {
+ struct cgroup_subsys_state css;
+ enum freezer_state state;
+ spinlock_t lock; /* protects _writes_ to state */
+};
+
```

```
+static inline struct freezer *cgroup_freezer(
+  struct cgroup *cgroup)
+{
+ return container_of(
+  cgroup_subsys_state(cgroup, freezer_subsys_id),
+  struct freezer, css);
+}
+
+static inline struct freezer *task_freezer(struct task_struct *task)
+{
+ return container_of(task_subsys_state(task, freezer_subsys_id),
+       struct freezer, css);
+}
+
+static inline int cgroup_frozen(struct task_struct *task)
+{
+ struct freezer *freezer;
+ enum freezer_state state;
+
+ task_lock(task);
+ freezer = task_freezer(task);
+ state = freezer->state;
+ task_unlock(task);
+
+ return state == STATE_FROZEN;
+}
+
+#else /* !CONFIG_CGROUP_FREEZER */
+
+static inline int cgroup_frozen(struct task_struct *task)
+{
+ return 0;
+}
+
+#endif /* !CONFIG_CGROUP_FREEZER */
+
+#endif /* _LINUX_CGROUP_FREEZER_H */
Index: linux-2.6.26-rc5-mm2/include/linux/cgroup_subsys.h
===================================================================
--- linux-2.6.26-rc5-mm2.orig/include/linux/cgroup_subsys.h
+++ linux-2.6.26-rc5-mm2/include/linux/cgroup_subsys.h
@@ -50,5 +50,11 @@ SUBSYS(devices)
 #ifdef CONFIG_CGROUP_MEMRLIMIT_CTLR
 SUBSYS(memrlimit_cgroup)
 #endif

 /* */
+
```

```
+#ifdef CONFIG_CGROUP_FREEZER
+SUBSYS(freezer)
+#endif
+
+/* */
Index: linux-2.6.26-rc5-mm2/init/Kconfig
===================================================================
--- linux-2.6.26-rc5-mm2.orig/init/Kconfig
+++ linux-2.6.26-rc5-mm2/init/Kconfig
@@ -329,10 +329,17 @@ config GROUP_SCHED
  default n
  help
    This feature lets CPU scheduler recognize task groups and control CPU
    bandwidth allocation to such task groups.

+config CGROUP_FREEZER
+       bool "control group freezer subsystem"
+       depends on CGROUPS
+       help
+         Provides a way to freeze and unfreeze all tasks in a
+   cgroup
+
 config FAIR_GROUP_SCHED
  bool "Group scheduling for SCHED_OTHER"
  depends on GROUP_SCHED
  default GROUP_SCHED

Index: linux-2.6.26-rc5-mm2/kernel/Makefile
===================================================================
--- linux-2.6.26-rc5-mm2.orig/kernel/Makefile
+++ linux-2.6.26-rc5-mm2/kernel/Makefile
@@ -49,10 +49,11 @@ obj-$(CONFIG_PM) += power/
 obj-$(CONFIG_BSD_PROCESS_ACCT) += acct.o
 obj-$(CONFIG_KEXEC) += kexec.o
 obj-$(CONFIG_COMPAT) += compat.o
 obj-$(CONFIG_CGROUPS) += cgroup.o
 obj-$(CONFIG_CGROUP_DEBUG) += cgroup_debug.o
+obj-$(CONFIG_CGROUP_FREEZER) += cgroup_freezer.o
 obj-$(CONFIG_CPUSETS) += cpuset.o
 obj-$(CONFIG_CGROUP_NS) += ns_cgroup.o
 obj-$(CONFIG_UTS_NS) += utsname.o
 obj-$(CONFIG_USER_NS) += user_namespace.o
 obj-$(CONFIG_PID_NS) += pid_namespace.o
Index: linux-2.6.26-rc5-mm2/kernel/cgroup_freezer.c
===================================================================
--- /dev/null
+++ linux-2.6.26-rc5-mm2/kernel/cgroup_freezer.c
@@ -0,0 +1,317 @@
```

```
+/*
+ * cgroup_freezer.c -  control group freezer subsystem
+ *
+ * Copyright IBM Corporation, 2007
+ *
+ * Author : Cedric Le Goater <clg@fr.ibm.com>
+ *
+ * This program is free software; you can redistribute it and/or modify it
+ * under the terms of version 2.1 of the GNU Lesser General Public License
+ * as published by the Free Software Foundation.
+ *
+ * This program is distributed in the hope that it would be useful, but
+ * WITHOUT ANY WARRANTY; without even the implied warranty of
+ * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
+ */
+
+#include <linux/module.h>
+#include <linux/cgroup.h>
+#include <linux/fs.h>
+#include <linux/uaccess.h>
+#include <linux/freezer.h>
+#include <linux/cgroup_freezer.h>
+#include <linux/seq_file.h>
+
+/*
+ * Buffer size for freezer state is limited by cgroups write_string()
+ * interface. See cgroups code for the current size.
+ */
+static const char *freezer_state_strs[] = {
+ "RUNNING",
+ "FREEZING",
+ "FROZEN",
+};
+
+/*
+ * State diagram (transition labels in parenthesis):
+ *
+ *  RUNNING -(FROZEN)-> FREEZING -(FROZEN)-> FROZEN
+ *    ^ ^                 |                  |
+ *    | |____(RUNNING)_____|                  |
+ *    |_____________________________(RUNNING)____|
+ */
+
+struct cgroup_subsys freezer_subsys;
+
+/* Locks taken and their ordering:
+ *
+ * freezer_create(), freezer_destroy():
```

```
+ * cgroup_lock [ by cgroup core ]
+ *
+ * can_attach():
+ * cgroup_lock
+ *
+ * cgroup_frozen():
+ * task_lock
+ *
+ * freezer_fork():
+ * task_lock
+ *  freezer->lock
+ *   sighand->siglock
+ *
+ * freezer_read():
+ * cgroup_lock
+ *  freezer->lock
+ *   read_lock css_set_lock
+ *
+ * freezer_write():
+ * cgroup_lock
+ *   freezer->lock
+ *    read_lock css_set_lock
+ *     [unfreeze: task_lock (reaquire freezer->lock)]
+ *      sighand->siglock
+ */
+static struct cgroup_subsys_state *freezer_create(struct cgroup_subsys *ss,
+        struct cgroup *cgroup)
+{
+ struct freezer *freezer;
+
+ freezer = kzalloc(sizeof(struct freezer), GFP_KERNEL);
+ if (!freezer)
+  return ERR_PTR(-ENOMEM);
+
+ spin_lock_init(&freezer->lock);
+ freezer->state = STATE_RUNNING;
+ return &freezer->css;
+}
+
+static void freezer_destroy(struct cgroup_subsys *ss,
+       struct cgroup *cgroup)
+{
+ kfree(cgroup_freezer(cgroup));
+}
+
+
+static int freezer_can_attach(struct cgroup_subsys *ss,
+         struct cgroup *new_cgroup,
```

```
+         struct task_struct *task)
+{
+ struct freezer *freezer;
+ int retval = 0;
+
+ /*
+  * The call to cgroup_lock() in the freezer.state write method prevents
+  * a write to that file racing against an attach, and hence the
+  * can_attach() result will remain valid until the attach completes.
+  */
+ freezer = cgroup_freezer(new_cgroup);
+ if (freezer->state == STATE_FROZEN)
+  retval = -EBUSY;
+ return retval;
+}
+
+static void freezer_fork(struct cgroup_subsys *ss, struct task_struct *task)
+{
+ struct freezer *freezer;
+
+ task_lock(task);
+ freezer = task_freezer(task);
+
+ BUG_ON(freezer->state == STATE_FROZEN);
+
+ /* Locking avoids race with FREEZING -> RUNNING transitions. */
+ spin_lock_irq(&freezer->lock);
+ if (freezer->state == STATE_FREEZING)
+  freeze_task(task, true);
+ spin_unlock_irq(&freezer->lock);
+
+ task_unlock(task);
+}
+
+/*
+ * caller must hold freezer->lock
+ */
+static void check_if_frozen(struct cgroup *cgroup,
+        struct freezer *freezer)
+{
+ struct cgroup_iter it;
+ struct task_struct *task;
+ unsigned int nfrozen = 0, ntotal = 0;
+
+ cgroup_iter_start(cgroup, &it);
+ while ((task = cgroup_iter_next(cgroup, &it))) {
+  ntotal++;
+  /*
```

```
+   * Task is frozen or will freeze immediately when next it gets
+   * woken
+   */
+  if (frozen(task) ||
+      (task_is_stopped_or_traced(task) && freezing(task)))
+   nfrozen++;
+ }
+
+ /*
+  * Transition to FROZEN when no new tasks can be added ensures
+  * that we never exist in the FROZEN state while there are unfrozen
+  * tasks.
+  */
+ if (nfrozen == ntotal)
+  freezer->state = STATE_FROZEN;
+ cgroup_iter_end(cgroup, &it);
+}
+
+static ssize_t freezer_read(struct cgroup *cgroup, struct cftype *cft,
+       struct seq_file *m)
+{
+ struct freezer *freezer;
+ enum freezer_state state;
+
+ if (!cgroup_lock_live_group(cgroup))
+  return -ENODEV;
+
+ freezer = cgroup_freezer(cgroup);
+ spin_lock_irq(&freezer->lock);
+ state = freezer->state;
+ if (state == STATE_FREEZING) {
+  /* We change from FREEZING to FROZEN lazily if the cgroup was
+   * only partially frozen when we exitted write. */
+  check_if_frozen(cgroup, freezer);
+  state = freezer->state;
+ }
+ spin_unlock_irq(&freezer->lock);
+ cgroup_unlock();
+
+ seq_puts(m, freezer_state_strs[state]);
+ seq_putc(m, '\n');
+ return 0;
+}
+
+static int try_to_freeze_cgroup(struct cgroup *cgroup, struct freezer *freezer)
+{
+ struct cgroup_iter it;
+ struct task_struct *task;
```

```
+ unsigned int num_cant_freeze_now = 0;
+
+ freezer->state = STATE_FREEZING;
+ cgroup_iter_start(cgroup, &it);
+ while ((task = cgroup_iter_next(cgroup, &it))) {
+  if (!freeze_task(task, true))
+   continue;
+  if (task_is_stopped_or_traced(task) && freezing(task))
+   /*
+    * The freeze flag is set so these tasks will
+    * immediately go into the fridge upon waking.
+    */
+   continue;
+  if (!freezing(task) && !freezer_should_skip(task))
+   num_cant_freeze_now++;
+ }
+ cgroup_iter_end(cgroup, &it);
+
+ return num_cant_freeze_now ? -EBUSY : 0;
+}
+
+static int unfreeze_cgroup(struct cgroup *cgroup, struct freezer *freezer)
+{
+ struct cgroup_iter it;
+ struct task_struct *task;
+
+ cgroup_iter_start(cgroup, &it);
+ while ((task = cgroup_iter_next(cgroup, &it))) {
+  int do_wake;
+
+  /* Drop freezer->lock to fix lock ordering (see freezer_fork) */
+  spin_unlock_irq(&freezer->lock);
+  task_lock(task);
+  spin_lock_irq(&freezer->lock);
+  do_wake = __thaw_process(task);
+  task_unlock(task);
+  if (do_wake)
+   wake_up_process(task);
+ }
+ cgroup_iter_end(cgroup, &it);
+ freezer->state = STATE_RUNNING;
+
+ return 0;
+}
+
+static int freezer_change_state(struct cgroup *cgroup,
+    enum freezer_state goal_state)
+{
```

```
+ struct freezer *freezer;
+ int retval = 0;
+
+ freezer = cgroup_freezer(cgroup);
+ spin_lock_irq(&freezer->lock);
+ check_if_frozen(cgroup, freezer); /* may update freezer->state */
+ if (goal_state == freezer->state)
+  goto out;
+ switch (freezer->state) {
+ case STATE_RUNNING:
+  retval = try_to_freeze_cgroup(cgroup, freezer);
+  break;
+ case STATE_FREEZING:
+  if (goal_state == STATE_FROZEN) {
+   /* Userspace is retrying after
+    * "/bin/echo FROZEN > freezer.state" returned -EBUSY */
+   retval = try_to_freeze_cgroup(cgroup, freezer);
+   break;
+  }
+  /* state == FREEZING and goal_state == RUNNING, so unfreeze */
+ case STATE_FROZEN:
+  retval = unfreeze_cgroup(cgroup, freezer);
+  break;
+ default:
+  break;
+ }
+out:
+ spin_unlock_irq(&freezer->lock);
+
+ return retval;
+}
+
+static int freezer_write(struct cgroup *cgroup,
+        struct cftype *cft,
+        const char *buffer)
+{
+ int retval;
+ enum freezer_state goal_state;
+
+ if (strcmp(buffer, freezer_state_strs[STATE_RUNNING]) == 0)
+  goal_state = STATE_RUNNING;
+ else if (strcmp(buffer, freezer_state_strs[STATE_FROZEN]) == 0)
+  goal_state = STATE_FROZEN;
+ else
+  return -EIO;
+
+ if (!cgroup_lock_live_group(cgroup))
+  return -ENODEV;
```

```
+ retval = freezer_change_state(cgroup, goal_state);
+ cgroup_unlock();
+ return retval;
+}
+
+static struct cftype files[] = {
+ {
+  .name = "state",
+  .read_seq_string = freezer_read,
+  .write_string = freezer_write,
+ },
+};
+
+static int freezer_populate(struct cgroup_subsys *ss, struct cgroup *cgroup)
+{
+ return cgroup_add_files(cgroup, ss, files, ARRAY_SIZE(files));
+}
+
+struct cgroup_subsys freezer_subsys = {
+ .name  = "freezer",
+ .create  = freezer_create,
+ .destroy = freezer_destroy,
+ .populate = freezer_populate,
+ .subsys_id = freezer_subsys_id,
+ .can_attach = freezer_can_attach,
+ .attach  = NULL,
+ .fork  = freezer_fork,
+ .exit  = NULL,
+};
Index: linux-2.6.26-rc5-mm2/include/linux/freezer.h
===================================================================
--- linux-2.6.26-rc5-mm2.orig/include/linux/freezer.h
+++ linux-2.6.26-rc5-mm2/include/linux/freezer.h
@@ -44,26 +44,34 @@ static inline bool should_send_signal(st
 }

 /*
  * Wake up a frozen process
  *
- * task_lock() is taken to prevent the race with refrigerator() which may
+ * task_lock() is needed to prevent the race with refrigerator() which may
  * occur if the freezing of tasks fails.  Namely, without the lock, if the
  * freezing of tasks failed, thaw_tasks() might have run before a task in
  * refrigerator() could call frozen_process(), in which case the task would be
  * frozen and no one would thaw it.
  */
-static inline int thaw_process(struct task_struct *p)
+static inline int __thaw_process(struct task_struct *p)
```

```
 {
- task_lock(p);
  if (frozen(p)) {
   p->flags &= ~PF_FROZEN;
+  return 1;
+ }
+ clear_freeze_flag(p);
+ return 0;
+}
+
+static inline int thaw_process(struct task_struct *p)
+{
+ task_lock(p);
+ if (__thaw_process(p) == 1) {
   task_unlock(p);
   wake_up_process(p);
   return 1;
  }
- clear_freeze_flag(p);
  task_unlock(p);
  return 0;
 }

 extern void refrigerator(void);

--
```

_____

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