Subject: Re: [PATCH] new cgroup controller "fork"
Posted by Paul Menage on Fri, 18 Feb 2011 00:59:51 GMT
View Forum Message <> Reply to Message

On Thu, Feb 17, 2011 at 5:31 AM, Max Kellermann <mk@cm4all.com> wrote:
> Can limit the number of fork()/clone() calls in a cgroup. It is
> useful as a safeguard against fork bombs.

I'd be inclined to simplify this a bit - avoid impacting the fork()
path twice, with cgroup_fork_pre_fork() and cgroup_fork_fork() and
just do the checks and decrements in a single pass. (In the event of
hitting a limit, you may need to make another partial pass up the tree
to restore the charged fork attempts)

Yes, it's true that you might charge for a fork() that later failed
for some other reason, but this will very rare (except on a machine
that's already screwed for other reasons) so that I don't think anyone
would complain about it. Especially if you explicitly document
"fork.remaining" as number of permitted "fork attempts".

Also, it would be slightly clearer to use fork_cgroup_* rather than
cgroup_fork_* - this makes it clearer that it's part of a cgroups
subsystem called "fork", rather than part of the cgroups core
framework.

I don't think that you need to make your spinlock IRQ-safe - AFAICS
nothing accesses it from the interrupt path.

Paul

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

Subject: Re: [PATCH] new cgroup controller "fork"
Posted by Max Kellermann on Fri, 18 Feb 2011 09:26:52 GMT
View Forum Message <> Reply to Message

On 2011/02/18 01:59, Paul Menage <menage@google.com> wrote:
> I'd be inclined to simplify this a bit - avoid impacting the fork()
> path twice, with cgroup_fork_pre_fork() and cgroup_fork_fork() and
> just do the checks and decrements in a single pass. (In the event of
> hitting a limit, you may need to make another partial pass up the tree
> to restore the charged fork attempts)

I have implemented it, but I don't like your idea. It actually
complicates the code. It tries to do two things at once, and running

again until it hits the failed cgroup seems somewhat fragile. I
believe the overhead for doing two separate runs in case of success is
negligible compared to the cost of sys_fork().

(Documentation not adjusted yet in the new patch)

> Also, it would be slightly clearer to use fork_cgroup_* rather than
> cgroup_fork_* - this makes it clearer that it's part of a cgroups
> subsystem called "fork", rather than part of the cgroups core
> framework.

Changed, but I've preserved the file name cgroup_fork.c. Do you want
me to change that, too? (What about cgroup_freezer.c and the config
option names CONFIG_CGROUP_*?)

> I don't think that you need to make your spinlock IRQ-safe - AFAICS
> nothing accesses it from the interrupt path.

Changed.

On 2011/02/17 14:50, KAMEZAWA Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com> wrote:
> How about -EAGAIN here ? I think it's not good to add new error code
> for system calls.

Changed that, but that got me a funny quirk while testing:

```
bear:~# ls
-bash: fork: retry: Resource temporarily unavailable
-bash: fork: retry: Resource temporarily unavailable
-bash: fork: retry: Resource temporarily unavailable
-bash: fork: retry: Resource temporarily unavailable
-bash: fork: Resource temporarily unavailable
bear:~#
```

Generally, I don't think EAGAIN is a good errno code for
"adminstrative limit exceeded". EAGAIN's meaning is "try again
later". Usually there is something like poll() to wait until the
resource is available - but a process cannot wait for the adminstrator
to raise the configured limits. You could blame that quirk on bash,
because it does not consider that divergent definition of EAGAIN for
fork()..

The changed patch follows for further discussion; I'll repost the
complete patch with description again once we agree that it's
finished.

Max

```
diff --git a/Documentation/cgroups/fork.txt b/Documentation/cgroups/fork.txt
new file mode 100644
index 0000000..dfbf291
--- /dev/null
+++ b/Documentation/cgroups/fork.txt
@@ -0,0 +1,30 @@
+The "fork" Controller
+---------------------
+
+The "fork" controller limits the number of times a new child process
+or thread can be created.  It maintains a per-group counter which gets
+decremented on each fork() / clone().  When the counter reaches zero,
+no process in the cgroup is allowed to create new child
+processes/threads, even if existing ones quit.
+
+This has been proven useful in a shared hosting environment.  A new
+temporary cgroup is created for each CGI process, and the maximum fork
+count is configured to a sensible value.  Since CGIs are expected to
+run for only a short time with predictable resource usage, this may be
+an appropriate tool to limit the damage that a freaked CGI can do.
+
+Initially, the counter is set to -1, which is a magic value for
+"disabled" - no limits are imposed on the processes in the group.  To
+set a new value, type (in the working directory of that control
+group):
+
+ echo 16 > fork.remaining
+
+This examples allows 16 forks in the control group.  0 means no
+further forks are allowed.  The limit may be lowered or increased or
+even disabled at any time by a process with write permissions to the
+attribute.
+
+To check if a fork is allowed, the controller walks the cgroup
+hierarchy up, and verifies all ancestors.  The counter of all
+ancestors is decreased.
diff --git a/include/linux/cgroup_fork.h b/include/linux/cgroup_fork.h
new file mode 100644
index 0000000..aef1dbd
--- /dev/null
+++ b/include/linux/cgroup_fork.h
@@ -0,0 +1,26 @@
+#ifndef _LINUX_CGROUP_FORK_H
+#define _LINUX_CGROUP_FORK_H
+
+#ifdef CONFIG_CGROUP_FORK
+
```

```
+/**
+ * Checks if another fork is allowed.  Call this before creating a new
+ * child process.
+ *
+ * @return 0 on success, a negative errno value if forking should be
+ * denied
+ */
+int
+fork_cgroup_pre_fork(void);
+
+#else /* !CONFIG_CGROUP_FORK */
+
+static inline int
+fork_cgroup_pre_fork(void)
+{
+ return 0;
+}
+
+#endif /* !CONFIG_CGROUP_FORK */
+
+#endif /* !_LINUX_CGROUP_FORK_H */
diff --git a/include/linux/cgroup_subsys.h b/include/linux/cgroup_subsys.h
index ccefff0..8ead7f9 100644
--- a/include/linux/cgroup_subsys.h
+++ b/include/linux/cgroup_subsys.h
@@ -66,3 +66,9 @@ SUBSYS(blkio)
 #endif

 /* */
+
+#ifdef CONFIG_CGROUP_FORK
+SUBSYS(fork)
+#endif
+
+/* */
diff --git a/init/Kconfig b/init/Kconfig
index 17e2cfb..ef53a85 100644
--- a/init/Kconfig
+++ b/init/Kconfig
@@ -596,6 +596,12 @@ config CGROUP_FREEZER
    Provides a way to freeze and unfreeze all tasks in a
    cgroup.

+config CGROUP_FORK
+ bool "fork controller for cgroups"
+ help
+   Limits the number of fork() calls in a cgroup.  An application
+   for this is to make a cgroup safe against fork bombs.
```

```
+
 config CGROUP_DEVICE
  bool "Device controller for cgroups"
  help
diff --git a/kernel/Makefile b/kernel/Makefile
index 353d3fe..b58cc01 100644
--- a/kernel/Makefile
+++ b/kernel/Makefile
@@ -61,6 +61,7 @@ obj-$(CONFIG_BACKTRACE_SELF_TEST) += backtracetest.o
 obj-$(CONFIG_COMPAT) += compat.o
 obj-$(CONFIG_CGROUPS) += cgroup.o
 obj-$(CONFIG_CGROUP_FREEZER) += cgroup_freezer.o
+obj-$(CONFIG_CGROUP_FORK) += cgroup_fork.o
 obj-$(CONFIG_CPUSETS) += cpuset.o
 obj-$(CONFIG_CGROUP_NS) += ns_cgroup.o
 obj-$(CONFIG_UTS_NS) += utsname.o
diff --git a/kernel/cgroup_fork.c b/kernel/cgroup_fork.c
new file mode 100644
index 0000000..e56b2c6
--- /dev/null
+++ b/kernel/cgroup_fork.c
@@ -0,0 +1,186 @@
+/*
+ *  A cgroup implementation which limits the number of fork() calls.
+ *
+ *  This file is subject to the terms and conditions of the GNU General Public
+ *  License.  See the file COPYING in the main directory of the Linux
+ *  distribution for more details.
+ */
+
+#include <linux/cgroup.h>
+#include <linux/cgroup_fork.h>
+#include <linux/slab.h>
+
+struct cgroup_fork {
+ struct cgroup_subsys_state css;
+
+ /** protect the "remaining" attribute */
+ spinlock_t lock;
+
+ /**
+  * The remaining number of forks allowed.  -1 is the magic
+  * value for "unlimited".
+  */
+ int remaining;
+};
+
+/**
```

```
+ * Get the #cgrou_fork instance of the specified #cgroup.
+ */
+static inline struct cgroup_fork *
+fork_cgroup_group(struct cgroup *cgroup)
+{
+ return container_of(cgroup_subsys_state(cgroup, fork_subsys_id),
+       struct cgroup_fork, css);
+}
+
+/**
+ * Get the #cgroup_fork instance of the specified task.
+ */
+static inline struct cgroup_fork *
+fork_cgroup_task(struct task_struct *task)
+{
+ return container_of(task_subsys_state(current_task, fork_subsys_id),
+       struct cgroup_fork, css);
+}
+
+/**
+ * Get the #cgroup_fork instance of the current task.
+ */
+static inline struct cgroup_fork *
+fork_cgroup_current(void)
+{
+ return fork_cgroup_task(current_task);
+}
+
+static struct cgroup_subsys_state *
+fork_cgroup_create(struct cgroup_subsys *ss, struct cgroup *cgroup)
+{
+ struct cgroup_fork *t = kzalloc(sizeof(*t), GFP_KERNEL);
+ if (!t)
+  return ERR_PTR(-ENOMEM);
+
+ spin_lock_init(&t->lock);
+
+ t->remaining = -1;
+
+ return &t->css;
+}
+
+static void
+fork_cgroup_destroy(struct cgroup_subsys *ss, struct cgroup *cgroup)
+{
+ struct cgroup_fork *t = fork_cgroup_group(cgroup);
+
+ kfree(t);
```

```
+}
+
+static s64
+fork_cgroup_remaining_read(struct cgroup *cgroup, struct cftype *cft)
+{
+ struct cgroup_fork *t = fork_cgroup_group(cgroup);
+ int value;
+
+ spin_lock(&t->lock);
+ value = t->remaining;
+ spin_unlock(&t->lock);
+
+ return value;
+}
+
+static int
+fork_cgroup_remaining_write(struct cgroup *cgroup, struct cftype *cft,
+       s64 value)
+{
+ struct cgroup_fork *t = fork_cgroup_group(cgroup);
+
+ if (value < -1 || value > (1L << 30))
+  return -EINVAL;
+
+ spin_lock(&t->lock);
+ t->remaining = (int)value;
+ spin_unlock(&t->lock);
+
+ return 0;
+}
+
+static const struct cftype fork_cgroup_files[] =  {
+ {
+  .name = "remaining",
+  .read_s64 = fork_cgroup_remaining_read,
+  .write_s64 = fork_cgroup_remaining_write,
+ },
+};
+
+static int
+fork_cgroup_populate(struct cgroup_subsys *ss, struct cgroup *cgroup)
+{
+ if (cgroup->parent == NULL)
+  /* cannot limit the root cgroup */
+  return 0;
+
+ return cgroup_add_files(cgroup, ss, fork_cgroup_files,
+    ARRAY_SIZE(fork_cgroup_files));
```

```
+}
+
+struct cgroup_subsys fork_subsys = {
+ .name = "fork",
+ .create = fork_cgroup_create,
+ .destroy = fork_cgroup_destroy,
+ .populate = fork_cgroup_populate,
+ .subsys_id = fork_subsys_id,
+};
+
+/**
+ * After a failure, restore the "remaining" counter in all cgroups
+ * from the task_current's one up to the failed one.
+ */
+static void
+fork_cgroup_restore(struct cgroup_fork *until_excluding)
+{
+ struct cgroup_fork *t;
+
+ for (t = fork_cgroup_current(); t != until_excluding;
+      t = fork_cgroup_group(t->css.cgroup->parent)) {
+  spin_lock(&t->lock);
+
+  if (t->remaining >= 0)
+   ++t->remaining;
+
+  spin_unlock(&t->lock);
+ }
+}
+
+int
+fork_cgroup_pre_fork(void)
+{
+ struct cgroup_fork *t;
+ int err = 0;
+
+ rcu_read_lock();
+
+ for (t = fork_cgroup_current(); t->css.cgroup->parent != NULL;
+      t = fork_cgroup_group(t->css.cgroup->parent)) {
+  spin_lock(&t->lock);
+
+  if (t->remaining > 0)
+   /* decrement the counter */
+   --t->remaining;
+  else if (t->remaining == 0) {
+   /* fork manpage: "[...] RLIMIT_NPROC resource
+      limit was encountered." - should be close
```

```
+      enough to this condition */
+    spin_unlock(&t->lock);
+    err = -EAGAIN;
+
+    /* restore the decremented counters */
+    fork_cgroup_restore(t);
+    break;
+  }
+
+  spin_unlock(&t->lock);
+ }
+
+ rcu_read_unlock();
+
+ return err;
+}
diff --git a/kernel/fork.c b/kernel/fork.c
index 25e4291..0f06202 100644
--- a/kernel/fork.c
+++ b/kernel/fork.c
@@ -32,6 +32,7 @@
 #include <linux/capability.h>
 #include <linux/cpu.h>
 #include <linux/cgroup.h>
+#include <linux/cgroup_fork.h>
 #include <linux/security.h>
 #include <linux/hugetlb.h>
 #include <linux/swap.h>
@@ -1024,6 +1025,10 @@ static struct task_struct *copy_process(unsigned long clone_flags,
     current->signal->flags & SIGNAL_UNKILLABLE)
   return ERR_PTR(-EINVAL);

+ retval = fork_cgroup_pre_fork();
+ if (retval)
+  goto fork_out;
+
  retval = security_task_create(clone_flags);
  if (retval)
   goto fork_out;
```