Subject: [RFC] cpuset update_cgroup_cpus_allowed
Posted by Paul Jackson on Mon, 15 Oct 2007 07:11:15 GMT
View Forum Message <> Reply to Message

Paul M, David R, others -- how does this look?

From: Paul Jackson <pj@sgi.com>

Update the per-task cpus_allowed of each task in a cgroup
whenever it has a cpuset whose 'cpus' mask changes.

The change to basing cpusets on the cgroup (aka container)
infrastructure broke an essential cpuset hack. The old cpuset
code had used the act of reattaching a task to its own cpuset
(writing its pid back into the same 'tasks' file it was already
in) to trigger the code that updates the cpus_allowed cpumask
in the task struct to the cpus_allowed cpumask dictated by that
tasks cpuset.

This was a hack to avoid having code in the main scheduler
code path that checked for changes in the cpus_allowed by each
tasks cpuset, which would have unacceptable performance impact
on the scheduler.

The cgroup code avoids calling the update callout if a task
is reattached to the cgroup it is already attached to do.
This turned reattaching a task to its own cpuset into a no-op,
making it impossible to change a tasks CPU placement by changing
the cpus_allowed of the cpuset containing that task.

The right thing to do would be to have the code that updates a
cpusets cpus_allowed walk through each task currently in that
cpuset and update the cpus_allowed in that tasks task_struct.

This change does that, adding code called from cpuset
update_cpumask() that updates the task_struct cpus_allowed of
each task in a cgroup whenever it has a cpuset whose 'cpus'
is changed.

Signed-off-by: Paul Jackson <pj@sgi.com>

This patch applies anywhere after:
cpusets-decrustify-cpuset-mask-update-code.patch

```
 Documentation/cpusets.txt |   23 +++++----------
 kernel/cpuset.c           |   68 ++++++++++++++++++++++++++++++++++++++++++-----
```

```
 kernel/sched.c          |    3 ++
 mm/pdflush.c            |    3 ++
 4 files changed, 76 insertions(+), 21 deletions(-)

--- 2.6.23-mm1.orig/kernel/cpuset.c 2007-10-14 22:24:56.268309633 -0700
+++ 2.6.23-mm1/kernel/cpuset.c 2007-10-14 22:34:52.645364388 -0700
@@ -677,6 +677,64 @@ done:
 }

 /*
+ * update_cgroup_cpus_allowed(cont, cpus)
+ *
+ * Keep looping over the tasks in cgroup 'cont', up to 'ntasks'
+ * tasks at a time, setting each task->cpus_allowed to 'cpus',
+ * until all tasks in the cgroup have that cpus_allowed setting.
+ *
+ * The 'set_cpus_allowed()' call cannot be made while holding the
+ * css_set_lock lock embedded in the cgroup_iter_* calls, so we stash
+ * some task pointers, in the tasks[] array on the stack, then drop
+ * that lock (cgroup_iter_end) before looping over the stashed tasks
+ * to update their cpus_allowed fields.
+ *
+ * Making the const 'ntasks' larger would use more stack space (bad),
+ * and reduce the number of cgroup_iter_start/cgroup_iter_end calls
+ * (good).  But perhaps more importantly, it could allow any bugs
+ * lurking in the 'need_repeat' looping logic to remain hidden longer.
+ * So keep ntasks rather small, to ensure any bugs in this loop logic
+ * are exposed quickly.
+ */
+static void update_cgroup_cpus_allowed(struct cgroup *cont, cpumask_t *cpus)
+{
+ int need_repeat = true;
+
+ while (need_repeat) {
+  struct cgroup_iter it;
+  const int ntasks = 10;
+  struct task_struct *tasks[ntasks];
+  struct task_struct **p, **q;
+
+  need_repeat = false;
+  p = tasks;
+
+  cgroup_iter_start(cont, &it);
+  while (1) {
+   struct task_struct *t;
+
+   t = cgroup_iter_next(cont, &it);
+   if (!t)
```

```
+    break;
+   if (cpus_equal(*cpus, t->cpus_allowed))
+    continue;
+   if (p == tasks + ntasks) {
+    need_repeat = true;
+    break;
+   }
+   get_task_struct(t);
+   *p++ = t;
+  }
+  cgroup_iter_end(cont, &it);
+
+  for (q = tasks; q < p; q++) {
+   set_cpus_allowed(*q, *cpus);
+   put_task_struct(*q);
+  }
+ }
+}
+
+/*
  * Call with manage_mutex held.  May take callback_mutex during call.
  */

@@ -684,7 +742,6 @@ static int update_cpumask(struct cpuset
 {
  struct cpuset trialcs;
  int retval;
- int cpus_changed, is_load_balanced;

  /* top_cpuset.cpus_allowed tracks cpu_online_map; it's read-only */
  if (cs == &top_cpuset)
@@ -713,16 +770,15 @@ static int update_cpumask(struct cpuset
  if (retval < 0)
   return retval;

- cpus_changed = !cpus_equal(cs->cpus_allowed, trialcs.cpus_allowed);
- is_load_balanced = is_sched_load_balance(&trialcs);
+ if (cpus_equal(cs->cpus_allowed, trialcs.cpus_allowed))
+  return 0;

  mutex_lock(&callback_mutex);
  cs->cpus_allowed = trialcs.cpus_allowed;
  mutex_unlock(&callback_mutex);

- if (cpus_changed && is_load_balanced)
-  rebuild_sched_domains();
-
+ update_cgroup_cpus_allowed(cs->css.cgroup, &cs->cpus_allowed);
```

```
+ rebuild_sched_domains();
  return 0;
 }

--- 2.6.23-mm1.orig/Documentation/cpusets.txt 2007-10-14 22:24:56.236309148 -0700
+++ 2.6.23-mm1/Documentation/cpusets.txt 2007-10-14 22:25:59.953276792 -0700
@@ -523,21 +523,14 @@ from one cpuset to another, then the ker
  memory placement, as above, the next time that the kernel attempts
  to allocate a page of memory for that task.

-If a cpuset has its CPUs modified, then each task using that
-cpuset does _not_ change its behavior automatically.  In order to
-minimize the impact on the critical scheduling code in the kernel,
-tasks will continue to use their prior CPU placement until they
-are rebound to their cpuset, by rewriting their pid to the 'tasks'
-file of their cpuset.  If a task had been bound to some subset of its
-cpuset using the sched_setaffinity() call, and if any of that subset
-is still allowed in its new cpuset settings, then the task will be
-restricted to the intersection of the CPUs it was allowed on before,
-and its new cpuset CPU placement.  If, on the other hand, there is
-no overlap between a tasks prior placement and its new cpuset CPU
-placement, then the task will be allowed to run on any CPU allowed
-in its new cpuset.  If a task is moved from one cpuset to another,
-its CPU placement is updated in the same way as if the tasks pid is
-rewritten to the 'tasks' file of its current cpuset.
+If a cpuset has its 'cpus' modified, then each task in that cpuset
+will have its allowed CPU placement changed immediately.  Similarly,
+if a tasks pid is written to a cpusets 'tasks' file, in either its
+current cpuset or another cpuset, then its allowed CPU placement is
+changed immediately.  If such a task had been bound to some subset
+of its cpuset using the sched_setaffinity() call, the task will be
+allowed to run on any CPU allowed in its new cpuset, negating the
+affect of the prior sched_setaffinity() call.

  In summary, the memory placement of a task whose cpuset is changed is
  updated by the kernel, on the next allocation of a page for that task,
--- 2.6.23-mm1.orig/kernel/sched.c 2007-10-14 22:24:56.340310725 -0700
+++ 2.6.23-mm1/kernel/sched.c 2007-10-14 22:25:59.973277096 -0700
@@ -51,6 +51,7 @@
 #include <linux/rcupdate.h>
 #include <linux/cpu.h>
 #include <linux/cpuset.h>
+#include <linux/cgroup.h>
 #include <linux/percpu.h>
 #include <linux/cpu_acct.h>
 #include <linux/kthread.h>
@@ -4335,9 +4336,11 @@ long sched_setaffinity(pid_t pid, cpumas
  if (retval)
```

```
   goto out_unlock;

+ cgroup_lock();
  cpus_allowed = cpuset_cpus_allowed(p);
  cpus_and(new_mask, new_mask, cpus_allowed);
  retval = set_cpus_allowed(p, new_mask);
+ cgroup_unlock();

 out_unlock:
  put_task_struct(p);
--- 2.6.23-mm1.orig/mm/pdflush.c 2007-10-14 22:23:28.710981177 -0700
+++ 2.6.23-mm1/mm/pdflush.c 2007-10-14 22:25:59.989277340 -0700
@@ -21,6 +21,7 @@
 #include <linux/writeback.h> // Prototypes pdflush_operation()
 #include <linux/kthread.h>
 #include <linux/cpuset.h>
+#include <linux/cgroup.h>
 #include <linux/freezer.h>


@@ -187,8 +188,10 @@ static int pdflush(void *dummy)
   * This is needed as pdflush's are dynamically created and destroyed.
   * The boottime pdflush's are easily placed w/o these 2 lines.
   */
+ cgroup_lock();
  cpus_allowed = cpuset_cpus_allowed(current);
  set_cpus_allowed(current, cpus_allowed);
+ cgroup_unlock();

  return __pdflush(&my_work);
 }

--
                        I won't rest till it's the best ...
                        Programmer, Linux Scalability
                        Paul Jackson <pj@sgi.com> 1.650.933.1373
```

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

---

Subject: Re: [RFC] cpuset update_cgroup_cpus_allowed
Posted by David Rientjes on Mon, 15 Oct 2007 18:49:02 GMT
View Forum Message <> Reply to Message

On Mon, 15 Oct 2007, Paul Jackson wrote:

```
> --- 2.6.23-mm1.orig/kernel/cpuset.c 2007-10-14 22:24:56.268309633 -0700
> +++ 2.6.23-mm1/kernel/cpuset.c 2007-10-14 22:34:52.645364388 -0700
> @@ -677,6 +677,64 @@ done:
>  }
>
>  /*
> + * update_cgroup_cpus_allowed(cont, cpus)
> + *
> + * Keep looping over the tasks in cgroup 'cont', up to 'ntasks'
> + * tasks at a time, setting each task->cpus_allowed to 'cpus',
> + * until all tasks in the cgroup have that cpus_allowed setting.
> + *
> + * The 'set_cpus_allowed()' call cannot be made while holding the
> + * css_set_lock lock embedded in the cgroup_iter_* calls, so we stash
> + * some task pointers, in the tasks[] array on the stack, then drop
> + * that lock (cgroup_iter_end) before looping over the stashed tasks
> + * to update their cpus_allowed fields.
> + *
> + * Making the const 'ntasks' larger would use more stack space (bad),
> + * and reduce the number of cgroup_iter_start/cgroup_iter_end calls
> + * (good).  But perhaps more importantly, it could allow any bugs
> + * lurking in the 'need_repeat' looping logic to remain hidden longer.
> + * So keep ntasks rather small, to ensure any bugs in this loop logic
> + * are exposed quickly.
> + */
> +static void update_cgroup_cpus_allowed(struct cgroup *cont, cpumask_t *cpus)
> +{
> + int need_repeat = true;
> +
> + while (need_repeat) {
> +  struct cgroup_iter it;
> +  const int ntasks = 10;
> +  struct task_struct *tasks[ntasks];
> +  struct task_struct **p, **q;
> +
> +  need_repeat = false;
> +  p = tasks;
> +
> +  cgroup_iter_start(cont, &it);
> +  while (1) {
> +   struct task_struct *t;
> +
> +   t = cgroup_iter_next(cont, &it);
> +   if (!t)
> +    break;
> +   if (cpus_equal(*cpus, t->cpus_allowed))
> +    continue;
```

By making this cpus_equal() and not cpus_intersects(), you're trying to make sure that t->cpus_allowed is always equal to *cpus for each task in the iterator.

```
> +   if (p == tasks + ntasks) {
> +    need_repeat = true;
> +    break;
> +   }
> +   get_task_struct(t);
> +   *p++ = t;
> +  }
> +  cgroup_iter_end(cont, &it);
> +
> +  for (q = tasks; q < p; q++) {
> +   set_cpus_allowed(*q, *cpus);
> +   put_task_struct(*q);
> +  }
> + }
> +}
```

Yet by not doing any locking here to prevent a cpu from being hot-unplugged, you can race and allow the hot-unplug event to happen before calling set_cpus_allowed(). That makes this entire function a no-op with set_cpus_allowed() returning -EINVAL for every call, which isn't caught, and no error is reported to userspace.

Now all the tasks in the cpuset have an inconsistent state with respect to their p->cpuset->cpus_allowed, because that was already updated in update_cpumask(). When userspace checks that value via the 'cpus' file, this is the value returned which is actually not true at all for any of the tasks in 'tasks'.

David

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

---

Subject: Re: [RFC] cpuset update_cgroup_cpus_allowed
Posted by Paul Menage on Mon, 15 Oct 2007 21:24:21 GMT
View Forum Message <> Reply to Message

Paul Jackson wrote:
> Paul M, David R, others -- how does this look?
>

Looks plausible, although as David comments I don't think it handles a concurrent CPU

hotplug/unplug. Also I don't like the idea of doing a cgroup_lock() across sched_setaffinity() - cgroup_lock() can be held for relatively long periods of time.

Here's an alternative for consideration, below. The main differences are:

- currently against an older kernel with pre-cgroup cpusets, so it uses tasklist_lock and do_each_thread(); a cgroup version would use cgroup iterators as yours does

- solves the race between sched_setaffinity() and update_cpumask() by having sched_setaffinity() check for changes to cpuset_cpus_allowed() after doing set_cpus_allowed()

- guarantees to only act on each process once (so guarantees forward progress, in the absence of fork bombs. (And could be adapted to handle fork bombs too)

- uses a priority heap to pick the processes to act on, based on start time

- uses lock_cpu_hotplug() to avoid races with CPU hotplug; sadly I think this is gone in more recent kernels, so some other synchronization would be needed

Paul

> From: Paul Jackson <pj@sgi.com>
>
> Update the per-task cpus_allowed of each task in a cgroup
> whenever it has a cpuset whose 'cpus' mask changes.
>
> The change to basing cpusets on the cgroup (aka container)
> infrastructure broke an essential cpuset hack. The old cpuset
> code had used the act of reattaching a task to its own cpuset
> (writing its pid back into the same 'tasks' file it was already
> in) to trigger the code that updates the cpus_allowed cpumask
> in the task struct to the cpus_allowed cpumask dictated by that
> tasks cpuset.
>
> This was a hack to avoid having code in the main scheduler
> code path that checked for changes in the cpus_allowed by each
> tasks cpuset, which would have unacceptable performance impact
> on the scheduler.
>
> The cgroup code avoids calling the update callout if a task
> is reattached to the cgroup it is already attached to do.
> This turned reattaching a task to its own cpuset into a no-op,
> making it impossible to change a tasks CPU placement by changing
> the cpus_allowed of the cpuset containing that task.
>
> The right thing to do would be to have the code that updates a
> cpusets cpus_allowed walk through each task currently in that
> cpuset and update the cpus_allowed in that tasks task_struct.

>
> This change does that, adding code called from cpuset
> update_cpumask() that updates the task_struct cpus_allowed of
> each task in a cgroup whenever it has a cpuset whose 'cpus'
> is changed.
>
> Signed-off-by: Paul Jackson <pj@sgi.com>
>
> ---
>
> This patch applies anywhere after:
>   cpusets-decrustify-cpuset-mask-update-code.patch
>
> Documentation/cpusets.txt | 23 +++++----------
> kernel/cpuset.c | 68 ++
> kernel/sched.c | 3 ++
> mm/pdflush.c | 3 ++
> 4 files changed, 76 insertions(+), 21 deletions(-)
>
> --- 2.6.23-mm1.orig/kernel/cpuset.c 2007-10-14 22:24:56.268309633 -0700
> +++ 2.6.23-mm1/kernel/cpuset.c 2007-10-14 22:34:52.645364388 -0700
> @@ -677,6 +677,64 @@ done:
> }
>
> /*
> + * update_cgroup_cpus_allowed(cont, cpus)
> + *
> + * Keep looping over the tasks in cgroup 'cont', up to 'ntasks'
> + * tasks at a time, setting each task->cpus_allowed to 'cpus',
> + * until all tasks in the cgroup have that cpus_allowed setting.
> + *
> + * The 'set_cpus_allowed()' call cannot be made while holding the
> + * css_set_lock lock embedded in the cgroup_iter_* calls, so we stash
> + * some task pointers, in the tasks[] array on the stack, then drop
> + * that lock (cgroup_iter_end) before looping over the stashed tasks
> + * to update their cpus_allowed fields.
> + *
> + * Making the const 'ntasks' larger would use more stack space (bad),
> + * and reduce the number of cgroup_iter_start/cgroup_iter_end calls
> + * (good). But perhaps more importantly, it could allow any bugs
> + * lurking in the 'need_repeat' looping logic to remain hidden longer.
> + * So keep ntasks rather small, to ensure any bugs in this loop logic
> + * are exposed quickly.
> + */
> +static void update_cgroup_cpus_allowed(struct cgroup *cont, cpumask_t *cpus)
> +{
> + int need_repeat = true;
> +

```
> + while (need_repeat) {
> +  struct cgroup_iter it;
> +  const int ntasks = 10;
> +  struct task_struct *tasks[ntasks];
> +  struct task_struct **p, **q;
> +
> +  need_repeat = false;
> +  p = tasks;
> +
> +  cgroup_iter_start(cont, &it);
> +  while (1) {
> +   struct task_struct *t;
> +
> +   t = cgroup_iter_next(cont, &it);
> +   if (!t)
> +    break;
> +   if (cpus_equal(*cpus, t->cpus_allowed))
> +    continue;
> +   if (p == tasks + ntasks) {
> +    need_repeat = true;
> +    break;
> +   }
> +   get_task_struct(t);
> +   *p++ = t;
> +  }
> +  cgroup_iter_end(cont, &it);
> +
> +  for (q = tasks; q < p; q++) {
> +   set_cpus_allowed(*q, *cpus);
> +   put_task_struct(*q);
> +  }
> + }
> +}
> +
> +/*
>   * Call with manage_mutex held.  May take callback_mutex during call.
>   */
>
> @@ -684,7 +742,6 @@ static int update_cpumask(struct cpuset
>  {
>   struct cpuset trialcs;
>   int retval;
> - int cpus_changed, is_load_balanced;
>
>   /* top_cpuset.cpus_allowed tracks cpu_online_map; it's read-only */
>   if (cs == &top_cpuset)
> @@ -713,16 +770,15 @@ static int update_cpumask(struct cpuset
>   if (retval < 0)
```

>   return retval;
>
> - cpus_changed = !cpus_equal(cs->cpus_allowed, trialcs.cpus_allowed);
> - is_load_balanced = is_sched_load_balance(&trialcs);
> + if (cpus_equal(cs->cpus_allowed, trialcs.cpus_allowed))
> + return 0;
>
>   mutex_lock(&callback_mutex);
>   cs->cpus_allowed = trialcs.cpus_allowed;
>   mutex_unlock(&callback_mutex);
>
> - if (cpus_changed && is_load_balanced)
> - rebuild_sched_domains();
> -
> + update_cgroup_cpus_allowed(cs->css.cgroup, &cs->cpus_allowed);
> + rebuild_sched_domains();
>   return 0;
> }
>
> --- 2.6.23-mm1.orig/Documentation/cpusets.txt 2007-10-14 22:24:56.236309148 -0700
> +++ 2.6.23-mm1/Documentation/cpusets.txt 2007-10-14 22:25:59.953276792 -0700
> @@ -523,21 +523,14 @@ from one cpuset to another, then the ker
> memory placement, as above, the next time that the kernel attempts
> to allocate a page of memory for that task.
>
> -If a cpuset has its CPUs modified, then each task using that
> -cpuset does _not_ change its behavior automatically. In order to
> -minimize the impact on the critical scheduling code in the kernel,
> -tasks will continue to use their prior CPU placement until they
> -are rebound to their cpuset, by rewriting their pid to the 'tasks'
> -file of their cpuset. If a task had been bound to some subset of its
> -cpuset using the sched_setaffinity() call, and if any of that subset
> -is still allowed in its new cpuset settings, then the task will be
> -restricted to the intersection of the CPUs it was allowed on before,
> -and its new cpuset CPU placement. If, on the other hand, there is
> -no overlap between a tasks prior placement and its new cpuset CPU
> -placement, then the task will be allowed to run on any CPU allowed
> -in its new cpuset. If a task is moved from one cpuset to another,
> -its CPU placement is updated in the same way as if the tasks pid is
> -rewritten to the 'tasks' file of its current cpuset.
> +If a cpuset has its 'cpus' modified, then each task in that cpuset
> +will have its allowed CPU placement changed immediately. Similarly,
> +if a tasks pid is written to a cpusets 'tasks' file, in either its#12 -
/usr/local/google/home/menage/kernel9/linux/kernel/cpuset.c ====
action=edit type=text
> +current cpuset or another cpuset, then its allowed CPU placement is
> +changed immediately. If such a task had been bound to some subset
> +of its cpuset using the sched_setaffinity() call, the task will be

> +allowed to run on any CPU allowed in its new cpuset, negating the
> +affect of the prior sched_setaffinity() call.
>
> In summary, the memory placement of a task whose cpuset is changed is
> updated by the kernel, on the next allocation of a page for that task,
> --- 2.6.23-mm1.orig/kernel/sched.c 2007-10-14 22:24:56.340310725 -0700
> +++ 2.6.23-mm1/kernel/sched.c 2007-10-14 22:25:59.973277096 -0700
> @@ -51,6 +51,7 @@
> #include <linux/rcupdate.h>
> #include <linux/cpu.h>
> #include <linux/cpuset.h>
> +#include <linux/cgroup.h>
> #include <linux/percpu.h>
> #include <linux/cpu_acct.h>
> #include <linux/kthread.h>
> @@ -4335,9 +4336,11 @@ long sched_setaffinity(pid_t pid, cpumas
>   if (retval)
>   goto out_unlock;
>
> + cgroup_lock();
>   cpus_allowed = cpuset_cpus_allowed(p);
>   cpus_and(new_mask, new_mask, cpus_allowed);
>   retval = set_cpus_allowed(p, new_mask);
> + cgroup_unlock();
>
> out_unlock:
>   put_task_struct(p);
> --- 2.6.23-mm1.orig/mm/pdflush.c 2007-10-14 22:23:28.710981177 -0700
> +++ 2.6.23-mm1/mm/pdflush.c 2007-10-14 22:25:59.989277340 -0700
> @@ -21,6 +21,7 @@
> #include <linux/writeback.h> // Prototypes pdflush_operation()
> #include <linux/kthread.h>
> #include <linux/cpuset.h>
> +#include <linux/cgroup.h>
> #include <linux/freezer.h>
>
>
> @@ -187,8 +188,10 @@ static int pdflush(void *dummy)
>   * This is needed as pdflush's are dynamically created and destroyed.
>   * The boottime pdflush's are easily placed w/o these 2 lines.
>   */
> + cgroup_lock();
>   cpus_allowed = cpuset_cpus_allowed(current);
>   set_cpus_allowed(current, cpus_allowed);
> + cgroup_unlock();
>
>   return __pdflush(&my_work);
> }

>

_____

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

---

Subject: Re: [RFC] cpuset update_cgroup_cpus_allowed
Posted by Paul Jackson on Tue, 16 Oct 2007 00:16:36 GMT
View Forum Message <> Reply to Message

Paul M wrote:
> Here's an alternative for consideration, below.

I don't see the alternative -- I just see my patch, with the added
blurbage:

 #12 - /usr/local/google/home/menage/kernel9/linux/kernel/cpuset.c ====
 # action=edit type=text

Should I be increasing my caffeine intake?

--
I won't rest till it's the best ...
Programmer, Linux Scalability
Paul Jackson <pj@sgi.com> 1.925.600.0401

_____

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

---

Subject: Re: [RFC] cpuset update_cgroup_cpus_allowed
Posted by Paul Menage on Tue, 16 Oct 2007 00:20:15 GMT
View Forum Message <> Reply to Message

Paul Jackson wrote:
> Paul M wrote:
>> Here's an alternative for consideration, below.
>
> I don't see the alternative -- I just see my patch, with the added
> blurbage:
>
> #12 - /usr/local/google/home/menage/kernel9/linux/kernel/cpuset.c ====
> # action=edit type=text
>

> Should I be increasing my caffeine intake?
>

Bah. Trying again:

Here's an alternative for consideration, below. The main differences are:

- currently against an older kernel with pre-cgroup cpusets, so it uses tasklist_lock and do_each_thread(); a cgroup version would use cgroup iterators as yours does

- solves the race between sched_setaffinity() and update_cpumask() by having sched_setaffinity() check for changes to cpuset_cpus_allowed() after doing set_cpus_allowed()

- guarantees to only act on each process once (so guarantees forward progress, in the absence of fork bombs. (And could be adapted to handle fork bombs too)

- uses a priority heap to pick the processes to act on, based on start time

- uses lock_cpu_hotplug() to avoid races with CPU hotplug; sadly I think this is gone in more recent kernels, so some other synchronization would be needed

Cause writes to cpuset "cpus" file to update cpus_allowed for member tasks:

- collect batches of tasks under tasklist_lock and then call set_cpus_allowed() on them outside the lock (since this can sleep).

- add a simple generic priority heap type to allow efficient collection of batches of tasks to be processed without duplicating or missing any tasks in subsequent batches.

- avoid races with hotplug events via lock_cpu_hotplug()

- make "cpus" file update a no-op if the mask hasn't changed

- fix race between update_cpumask() and sched_setaffinity() by making sched_setaffinity() to post-check that it's not running on any cpus outside cpuset_cpus_allowed().

```
include/linux/prio_heap.h |   56 ++++++++++++++++++++++++
kernel/cpuset.c           |  103 +++++++++++++++++++++++++++++++++++++++++++--
kernel/sched.c            |   13 +++++
```

```
 lib/Makefile            |    2
 lib/prio_heap.c         |   68 ++++++++++++++++++++++++++++++++++++++++++++++++
 5 files changed, 238 insertions(+), 4 deletions(-)
--- /dev/null 1969-12-31 16:00:00.000000000 -0800
+++ linux/include/linux/prio_heap.h 2007-10-12 16:43:27.000000000 -0700
@@ -0,0 +1,56 @@
+#ifndef _LINUX_PRIO_HEAP_H
+#define _LINUX_PRIO_HEAP_H
+
+/*
+ * Simple insertion-only static-sized priority heap containing
+ * pointers, based on CLR, chapter 7
+ */
+
+#include <linux/gfp.h>
+
+/**
+ * struct ptr_heap - simple static-sized priority heap
+ * @ptrs - pointer to data area
+ * @max - max number of elements that can be stored in @ptrs
+ * @size - current number of valid elements in @ptrs (in the range 0..@size-1
+ */
+struct ptr_heap {
+ void **ptrs;
+ int max;
+ int size;
+};
+
+/**
+ * heap_init - initialize an empty heap with a given memory size
+ * @heap: the heap structure to be initialized
+ * @size: amount of memory to use in bytes
+ * @gfp_mask: mask to pass to kmalloc()
+ */
+extern int heap_init(struct ptr_heap *heap, size_t size, gfp_t gfp_mask);
+
+/**
+ * heap_free - release a heap's storage
+ * @heap: the heap structure whose data should be released
+ */
+void heap_free(struct ptr_heap *heap);
+
+/**
+ * heap_insert - insert a value into the heap and return any overflowed value
+ * @heap: the heap to be operated on
+ * @p: the pointer to be inserted
+ * @gt: comparison operator, which should implement "greater than"
+ *
```

```
+ * Attempts to insert the given value into the priority heap. If the
+ * heap is full prior to the insertion, then the resulting heap will
+ * consist of the smallest @max elements of the original heap and the
+ * new element; the greatest element will be removed from the heap and
+ * returned. Note that the returned element will be the new element
+ * (i.e. no change to the heap) if the new element is greater than all
+ * elements currently in the heap.
+ */
+extern void *heap_insert(struct ptr_heap *heap, void *p,
+    int (*gt)(void *, void *));
+
+
+
+#endif /* _LINUX_PRIO_HEAP_H */
==== linux/kernel/cpuset.c
--- linux/kernel/cpuset.c 2007-10-05 17:46:09.000000000 -0700
+++ linux/kernel/cpuset.c 2007-10-12 16:24:49.000000000 -0700
@@ -37,6 +37,7 @@
 #include <linux/mount.h>
 #include <linux/namei.h>
 #include <linux/pagemap.h>
+#include <linux/prio_heap.h>
 #include <linux/proc_fs.h>
 #include <linux/rcupdate.h>
 #include <linux/sched.h>
@@ -839,6 +840,36 @@
  unlock_cpu_hotplug();
 }

+static int inline started_after_time(struct task_struct *t1,
+         struct timespec *time,
+         struct task_struct *t2)
+{
+ int start_diff = timespec_compare(&t1->start_time, time);
+ if (start_diff > 0) {
+  return 1;
+ } else if (start_diff < 0) {
+  return 0;
+ } else {
+  /*
+   * Arbitrarily, if two processes started at the same
+   * time, we'll say that the lower pointer value
+   * started first. Note that t2 may have exited by now
+   * so this may not be a valid pointer any longer, but
+   * that's fine - it still serves to distinguish
+   * between two tasks started (effectively)
+   * simultaneously.
+   */
```

```
+  return t1 > t2;
+ }
+}
+
+static int inline started_after(void *p1, void *p2)
+{
+ struct task_struct *t1 = p1;
+ struct task_struct *t2 = p2;
+ return started_after_time(t1, &t2->start_time, t2);
+}
+
 /*
  * Call with manage_mutex held.  May take callback_mutex during call.
  */
@@ -846,7 +877,12 @@
 static int update_cpumask(struct cpuset *cs, char *buf)
 {
  struct cpuset trialcs;
- int retval, cpus_unchanged;
+ int retval, i;
+ struct task_struct *g, *p, *dropped;
+ /* Never dereference latest_task, since it's not refcounted */
+ struct task_struct *latest_task = NULL;
+ struct ptr_heap heap;
+ struct timespec latest_time = { 0, 0 };

  /* top_cpuset.cpus_allowed tracks cpu_online_map; it's read-only */
  if (cs == &top_cpuset)
@@ -862,11 +898,72 @@
  retval = validate_change(cs, &trialcs);
  if (retval < 0)
   return retval;
- cpus_unchanged = cpus_equal(cs->cpus_allowed, trialcs.cpus_allowed);
+ if (cpus_equal(cs->cpus_allowed, trialcs.cpus_allowed))
+  return 0;
+ retval = heap_init(&heap, PAGE_SIZE, GFP_KERNEL);
+ if (retval)
+  return retval;
+
  mutex_lock(&callback_mutex);
  cs->cpus_allowed = trialcs.cpus_allowed;
  mutex_unlock(&callback_mutex);
- if (is_cpu_exclusive(cs) && !cpus_unchanged)
+ again:
+ read_lock(&tasklist_lock);
+ /*
+  * Scan tasks in the cpuset, and update the cpumasks of any
+  * that need an update. Since we can't call set_cpus_allowed()
```

```
+  * while holding tasklist_lock, gather tasks to be processed
+  * in a heap structure. If the statically-sized heap fills up,
+  * overflow tasks that started later, and in future iterations
+  * only consider tasks that started after the latest task in
+  * the previous pass. This guarantees forward progress and
+  * that we don't miss any tasks
+  */
+ heap.size = 0;
+ do_each_thread(g, p) {
+  /* Only affect tasks from this cpuset */
+  if (p->cpuset != cs)
+   continue;
+  /* Only affect tasks that don't have the right cpus_allowed */
+  if (cpus_equal(p->cpus_allowed, cs->cpus_allowed))
+   continue;
+  /*
+   * Only process tasks that started after the last task
+   * we processed
+   */
+  if (!started_after_time(p, &latest_time, latest_task))
+   continue;
+  dropped = heap_insert(&heap, p, &started_after);
+  if (dropped == NULL) {
+   get_task_struct(p);
+  } else if (dropped != p) {
+   get_task_struct(p);
+   put_task_struct(dropped);
+  }
+ } while_each_thread(g, p);
+ read_unlock(&tasklist_lock);
+ if (heap.size) {
+  for (i = 0; i < heap.size; i++) {
+   struct task_struct *p = heap.ptrs[i];
+   if (i == 0) {
+    latest_time = p->start_time;
+    latest_task = p;
+   }
+   set_cpus_allowed(p, cs->cpus_allowed);
+   put_task_struct(p);
+  }
+  /*
+   * If we had to process any tasks at all, scan again
+   * in case some of them were in the middle of forking
+   * children that didn't notice the new cpumask
+   * restriction.  Not the most efficient way to do it,
+   * but it avoids having to take callback_mutex in the
+   * fork path
+   */
```

```
+  goto again;
+ }
+ heap_free(&heap);
+ if (is_cpu_exclusive(cs))
   update_cpu_domains(cs);
  return 0;
 }
==== linux/kernel/sched.c
--- linux/kernel/sched.c 2007-10-11 20:07:17.000000000 -0700
+++ linux/kernel/sched.c 2007-10-11 22:04:45.000000000 -0700
@@ -4411,8 +4411,21 @@

  cpus_allowed = cpuset_cpus_allowed(p);
  cpus_and(new_mask, new_mask, cpus_allowed);
+ again:
  retval = set_cpus_allowed(p, new_mask);

+ if (!retval) {
+  cpus_allowed = cpuset_cpus_allowed(p);
+  if (!cpus_subset(new_mask, cpus_allowed)) {
+   /*
+    * We must have raced with a concurrent cpuset
+    * update. Just reset the cpus_allowed to the
+    * cpuset's cpus_allowed
+    */
+   new_mask = cpus_allowed;
+   goto again;
+  }
+ }
 out_unlock:
  put_task_struct(p);
  unlock_cpu_hotplug();
==== linux/lib/Makefile
--- linux/lib/Makefile 2007-10-15 14:09:45.000000000 -0700
+++ linux/lib/Makefile 2007-10-12 16:29:22.000000000 -0700
@@ -5,7 +5,7 @@
 lib-y := errno.o ctype.o string.o vsprintf.o cmdline.o \
   bust_spinlocks.o rbtree.o radix-tree.o dump_stack.o \
   idr.o div64.o int_sqrt.o bitmap.o extable.o prio_tree.o \
-  sha1.o
+  sha1.o prio_heap.o

 lib-$(CONFIG_SMP) += cpumask.o

==== linux/lib/prio_heap.c
--- /dev/null 1969-12-31 16:00:00.000000000 -0800
+++ linux/lib/prio_heap.c 2007-10-12 16:30:27.000000000 -0700
@@ -0,0 +1,68 @@
```

```
+/*
+ * Simple insertion-only static-sized priority heap containing
+ * pointers, based on CLR, chapter 7
+ */
+
+#include <linux/slab.h>
+#include <linux/prio_heap.h>
+
+int heap_init(struct ptr_heap *heap, size_t size, gfp_t gfp_mask)
+{
+ heap->ptrs = kmalloc(size, gfp_mask);
+ if (!heap->ptrs)
+  return -ENOMEM;
+ heap->size = 0;
+ heap->max = size / sizeof(void *);
+ return 0;
+}
+
+void heap_free(struct ptr_heap *heap)
+{
+ kfree(heap->ptrs);
+}
+
+void *heap_insert(struct ptr_heap *heap, void *p, int (*gt)(void *, void *))
+{
+ void *res;
+ void **ptrs = heap->ptrs;
+ int pos;
+
+ if (heap->size < heap->max) {
+  /* Heap insertion */
+  int pos = heap->size++;
+  while (pos > 0 && gt(p, ptrs[(pos-1)/2])) {
+   ptrs[pos] = ptrs[(pos-1)/2];
+   pos = (pos-1)/2;
+  }
+  ptrs[pos] = p;
+  return NULL;
+ }
+
+ /* The heap is full, so something will have to be dropped */
+
+ /* If the new pointer is greater than the current max, drop it */
+ if (gt(p, ptrs[0]))
+  return p;
+
+ /* Replace the current max and heapify */
+ res = ptrs[0];
```

```
+ ptrs[0] = p;
+ pos = 0;
+
+ while (1) {
+  int left = 2 * pos + 1;
+  int right = 2 * pos + 2;
+  int largest = pos;
+  if (left < heap->size && gt(ptrs[left], p))
+   largest = left;
+  if (right < heap->size && gt(ptrs[right], ptrs[largest]))
+   largest = right;
+  if (largest == pos)
+   break;
+  /* Push p down the heap one level and bump one up */
+  ptrs[pos] = ptrs[largest];
+  ptrs[largest] = p;
+  pos = largest;
+ }
+ return res;
+}
```

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

Subject: Re: [RFC] cpuset update_cgroup_cpus_allowed
Posted by Paul Jackson on Tue, 16 Oct 2007 02:32:59 GMT
View Forum Message <> Reply to Message

> Yet by not doing any locking here to prevent a cpu from being
> hot-unplugged, you can race and allow the hot-unplug event to happen
> before calling set_cpus_allowed(). That makes this entire function a
> no-op with set_cpus_allowed() returning -EINVAL for every call, which
> isn't caught, and no error is reported to userspace.

Good point ... hmmm ...

--
I won't rest till it's the best ...
Programmer, Linux Scalability
Paul Jackson <pj@sgi.com> 1.925.600.0401

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

Subject: Re: [RFC] cpuset update_cgroup_cpus_allowed
Posted by Paul Jackson on Tue, 16 Oct 2007 02:34:39 GMT
View Forum Message <> Reply to Message

> currently against an older kernel

ah .. which older kernel?

I tried it against the broken out 2.6.23-rc8-mm2 patch set,
inserting it before the task-containersv11-* patches, but
that blew up on me - three rejected hunks.

Any chance of getting this against a current cgroup (aka
container) kernel?

Could you use the diff --show-c-function option when composing
patches - they're easier to read that way - thanks.

```
+ if (!retval) {
+  cpus_allowed = cpuset_cpus_allowed(p);
+  if (!cpus_subset(new_mask, cpus_allowed)) {
+   /*
+    * We must have raced with a concurrent cpuset
+    * update. Just reset the cpus_allowed to the
+    * cpuset's cpus_allowed
+    */
+   new_mask = cpus_allowed;
```

This narrows the race, perhaps sufficiently, but I don't see that it
guarantees closure. Memory accesses to two different locations are not
guaranteed to be ordered across nodes, as best I recall. The second
line above, that rereads the cpuset cpus_allowed, could get an old
value, in essence.

```
 cpuset update task  sched_setaffinity task
 ------------------  ----------------------

 A. write cpuset [Q]  V. read cpuset [Q]
 B. read task [P]  W. check ok
 C. write task [P]  X. write task [P]
    Y. reread cpuset [Q]
    Z. check ok again
```

Two memory locations:
 [P] the cpus_allowed mask in the task_struct of the
  task doing the sched_setaffinity call.
 [Q] the cpus_allowed mask in the cpuset of the cpuset
  to which the sched_setaffinity task is attached.

Even though, from the perspective of location [P], both B. and C.
happened before X., still from the perspective of location [Q] the
rereading in Y. could return the value the cpuset cpus_allowed had
before the write in A. This could result in a task running with
a cpus_allowed that was totally outside its cpusets cpus_allowed.

I will grant that this is a narrow window. I won't loose much sleep
over it.

> - uses a priority heap to pick the processes to act on, based on start time

This adds a fair bit of code and complexity, relative to my patch.
This I do loose more sleep over. There has to be a compelling
reason for doing this.

The point that David raises, regarding the interaction of this with
hotplug, seems to be a compelling reason for doing -something-
different than my patch proposal.

I don't know yet if it compels us to this much code, however.

Any chance you could provide a patch that works against cgroups?

--
                I won't rest till it's the best ...
                Programmer, Linux Scalability
                Paul Jackson <pj@sgi.com> 1.925.600.0401

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

---

Subject: Re: [RFC] cpuset update_cgroup_cpus_allowed
Posted by Paul Menage on Tue, 16 Oct 2007 05:12:32 GMT
View Forum Message <> Reply to Message

On 10/15/07, Paul Jackson <pj@sgi.com> wrote:
> > currently against an older kernel
>
> ah .. which older kernel?

2.6.18, but I can do a version against 2.6.23-mm1.

> +	if (!retval) {
> +		cpus_allowed = cpuset_cpus_allowed(p);
> +		if (!cpus_subset(new_mask, cpus_allowed)) {
> +			/*

> +   * We must have raced with a concurrent cpuset
> +   * update. Just reset the cpus_allowed to the
> +   * cpuset's cpus_allowed
> +   */
> +   new_mask = cpus_allowed;
>
> This narrows the race, perhaps sufficiently, but I don't see that it
> guarantees closure. Memory accesses to two different locations are not
> guaranteed to be ordered across nodes, as best I recall. The second
> line above, that rereads the cpuset cpus_allowed, could get an old
> value, in essence.
>
>   cpuset update task   sched_setaffinity task
>   ------------------   ----------------------
>
>   A. write cpuset [Q]   V. read cpuset [Q]
>   B. read task [P]   W. check ok
>   C. write task [P]   X. write task [P]
>   Y. reread cpuset [Q]
>   Z. check ok again
>
> Two memory locations:
>   [P] the cpus_allowed mask in the task_struct of the
>   task doing the sched_setaffinity call.
>   [Q] the cpus_allowed mask in the cpuset of the cpuset
>   to which the sched_setaffinity task is attached.
>
> Even though, from the perspective of location [P], both B. and C.
> happened before X., still from the perspective of location [Q] the
> rereading in Y. could return the value the cpuset cpus_allowed had
> before the write in A. This could result in a task running with
> a cpus_allowed that was totally outside its cpusets cpus_allowed.

But cpuset_cpus_allowed() synchronizes on callback_mutex. So I assert
this race isn't an issue.

>
> I will grant that this is a narrow window. I won't loose much sleep
> over it.
>
> > - uses a priority heap to pick the processes to act on, based on start time
>
> This adds a fair bit of code and complexity, relative to my patch.
> This I do loose more sleep over. There has to be a compelling
> reason for doing this.

My plan was to hide this inside cgroup_iter_* so that users didn't
have to hold the cssgroup_lock across the entire iteration.

>
> The point that David raises, regarding the interaction of this with
> hotplug, seems to be a compelling reason for doing -something-
> different than my patch proposal.
>
> I don't know yet if it compels us to this much code, however.
>
> Any chance you could provide a patch that works against cgroups?
>

Will do - I justed wanted to get this quickly out to show the idea
that I was working on.

Paul

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

---

Subject: Re: [RFC] cpuset update_cgroup_cpus_allowed
Posted by Paul Jackson on Tue, 16 Oct 2007 05:20:17 GMT
View Forum Message <> Reply to Message

> Will do - I justed wanted to get this quickly out to show the idea
> that I was working on.

Ok - good.

In the final analysis, I'll take whatever works ;).

I'll lobby for keeping the code "simple" (a subjective metric) and poke
what holes I can in things, and propose what alternatives I can muster.

But so long as setting a cpusets 'cpus' in 2.6.24 leads, whether by my
historical "rewrite the pid to its own 'tasks' file" hack, or by a
proper solution such as you have advocated, or by some other scheme
or hack, to updating the cpus_allowed of each task in that cpuset, then
I'm ok.

Right now, that goal is not met, with the cgroup patches lined up in
*-mm for what will become 2.6.24.

We're getting short of time to fix this.

--
I won't rest till it's the best ...

Programmer, Linux Scalability
Paul Jackson <pj@sgi.com> 1.925.600.0401

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

---

Subject: Re: [RFC] cpuset update_cgroup_cpus_allowed
Posted by Paul Jackson on Tue, 16 Oct 2007 06:07:29 GMT
View Forum Message <> Reply to Message

> > + if (cpus_equal(*cpus, t->cpus_allowed))
> > + continue;
> > ...
> > + for (q = tasks; q < p; q++) {
> > + set_cpus_allowed(*q, *cpus);
> > + put_task_struct(*q);
> > + }
> > + }
> > +}
>
> Yet by not doing any locking here to prevent a cpu from being
> hot-unplugged, you can race and allow the hot-unplug event to happen
> before calling set_cpus_allowed(). That makes this entire function a
> no-op with set_cpus_allowed() returning -EINVAL for every call, which
> isn't caught, and no error is reported to userspace.
>
> Now all the tasks in the cpuset have an inconsistent state with respect to
> their p->cpuset->cpus_allowed, because that was already updated in
> update_cpumask().

My solution may be worse than that. Because set_cpus_allowed() will
fail if asked to set a non-overlapping cpumask, my solution could never
terminate. If asked to set a cpusets cpus to something that went off
line right then, this I'd guess this code could keep looping forever,
looking for cpumasks that didn't match, and then not noticing that it
was failing to set them so as they would match.

... it needs work ... or the alternative solution from Paul M.

--
I won't rest till it's the best ...
Programmer, Linux Scalability
Paul Jackson <pj@sgi.com> 1.925.600.0401

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

Subject: Re: [RFC] cpuset update_cgroup_cpus_allowed
Posted by David Rientjes on Tue, 16 Oct 2007 06:21:08 GMT
View Forum Message <> Reply to Message

On Mon, 15 Oct 2007, Paul Jackson wrote:

> My solution may be worse than that. Because set_cpus_allowed() will
> fail if asked to set a non-overlapping cpumask, my solution could never
> terminate. If asked to set a cpusets cpus to something that went off
> line right then, this I'd guess this code could keep looping forever,
> looking for cpumasks that didn't match, and then not noticing that it
> was failing to set them so as they would match.
>

Why can't you just add a helper function to sched.c:

```
 void set_hotcpus_allowed(struct task_struct *task,
     cpumask_t cpumask)
 {
  mutex_lock(&sched_hotcpu_mutex);
  set_cpus_allowed(task, cpumask);
  mutex_unlock(&sched_hotcpu_mutex);
 }
```

And then change each task's cpus_allowed via that function instead of
set_cpus_allowed() directly?

You don't need to worry about making the task->cpuset->cpus_allowed
assignment a critical section because common_cpu_mem_hotplug_unplug() will
remove any hot-unplugged cpus from each cpuset's cpus_allowed in the
hierarchy.

Your loop will still need to be reworked so that cgroup_iter_{start,end}()
are not reinvoked unnecessarily and you rely only on cgroup_iter_next()
returning NULL to determine when you've gone through the entire list.
There's no need to go back and check the cpus_allowed of tasks you've
already called set_cpus_allowed() on either directly or indirectly via my
helper function above.

  David

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

Subject: Re: [RFC] cpuset update_cgroup_cpus_allowed
Posted by Paul Jackson on Tue, 16 Oct 2007 09:16:26 GMT
View Forum Message <> Reply to Message

David wrote:
> Why can't you just add a helper function to sched.c:
>
> void set_hotcpus_allowed(struct task_struct *task,
>   cpumask_t cpumask)
> {
>   mutex_lock(&sched_hotcpu_mutex);
>   set_cpus_allowed(task, cpumask);
>   mutex_unlock(&sched_hotcpu_mutex);
> }
>
> And then change each task's cpus_allowed via that function instead of
> set_cpus_allowed() directly?

I guess this would avoid race conditions within the set_cpus_allowed()
routine, between its code to read the cpu_online_map and set the tasks
cpus_allowed ... though if that's useful, don't we really need to add
locking/unlocking on sched_hotcpu_mutex right inside the
set_cpus_allowed() routine, for all users of set_cpus_allowed ??

But I don't see where the above code helps at all deal with the
races I considered in my previous message:

> My solution may be worse than that. Because set_cpus_allowed() will
> fail if asked to set a non-overlapping cpumask, my solution could never
> terminate. If asked to set a cpusets cpus to something that went off
> line right then, this I'd guess this code could keep looping forever,
> looking for cpumasks that didn't match, and then not noticing that it
> was failing to set them so as they would match.

These races involve reading the tasks cpuset cpus_allowed mask, reading
the online map, and both reading and writing the tasks task_struct
cpus_allowed. Unless one holds the relevant lock for the entire
interval surrounding the critical accesses to these values, it won't do
any good that I can see. Just briefly holding a lock around each
separate access is useless.

--

                  I won't rest till it's the best ...
                  Programmer, Linux Scalability
                  Paul Jackson <pj@sgi.com> 1.925.600.0401

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

Subject: Re: [RFC] cpuset update_cgroup_cpus_allowed
Posted by Paul Menage on Tue, 16 Oct 2007 10:07:59 GMT
View Forum Message <> Reply to Message

Paul Jackson wrote:
>
> Any chance you could provide a patch that works against cgroups?
>

Fix cpusets update_cpumask

Cause writes to cpuset "cpus" file to update cpus_allowed for member
tasks:

- collect batches of tasks under tasklist_lock and then call
set_cpus_allowed() on them outside the lock (since this can sleep).

- add a simple generic priority heap type to allow efficient
collection of batches of tasks to be processed without
duplicating or missing any tasks in subsequent batches.

- make "cpus" file update a no-op if the mask hasn't changed

- fix race between update_cpumask() and sched_setaffinity() by making
sched_setaffinity() post-check that it's not running on any cpus
outside cpuset_cpus_allowed().

```
 include/linux/prio_heap.h |   56 ++++++++++++++++++++++++
 kernel/cpuset.c           |  103 +++++++++++++++++++++++++++++++++++++++++++--
 kernel/sched.c            |   13 +++++
 lib/Makefile              |    2
 include/linux/prio_heap.h |   58 ++++++++++++++++++++++++
 kernel/cpuset.c           |  105 +++++++++++++++++++++++++++++++++++++++++++--
 kernel/sched.c            |   13 +++++
 lib/Makefile              |    2
 lib/prio_heap.c           |   70 ++++++++++++++++++++++++++++++
 5 files changed, 243 insertions(+), 5 deletions(-)
Index: container-2.6.23-mm1/include/linux/prio_heap.h
=====================================================================
--- /dev/null
+++ container-2.6.23-mm1/include/linux/prio_heap.h
@@ -0,0 +1,58 @@
+#ifndef _LINUX_PRIO_HEAP_H
+#define _LINUX_PRIO_HEAP_H
+
+/*
```

```
+ * Simple insertion-only static-sized priority heap containing
+ * pointers, based on CLR, chapter 7
+ */
+
+#include <linux/gfp.h>
+
+/**
+ * struct ptr_heap - simple static-sized priority heap
+ * @ptrs - pointer to data area
+ * @max - max number of elements that can be stored in @ptrs
+ * @size - current number of valid elements in @ptrs (in the range 0..@size-1
+ * @gt: comparison operator, which should implement "greater than"
+ */
+struct ptr_heap {
+ void **ptrs;
+ int max;
+ int size;
+ int (*gt)(void *, void *);
+};
+
+/**
+ * heap_init - initialize an empty heap with a given memory size
+ * @heap: the heap structure to be initialized
+ * @size: amount of memory to use in bytes
+ * @gfp_mask: mask to pass to kmalloc()
+ * @gt: comparison operator, which should implement "greater than"
+ */
+extern int heap_init(struct ptr_heap *heap, size_t size, gfp_t gfp_mask,
+       int (*gt)(void *, void *));
+
+/**
+ * heap_free - release a heap's storage
+ * @heap: the heap structure whose data should be released
+ */
+void heap_free(struct ptr_heap *heap);
+
+/**
+ * heap_insert - insert a value into the heap and return any overflowed value
+ * @heap: the heap to be operated on
+ * @p: the pointer to be inserted
+ *
+ * Attempts to insert the given value into the priority heap. If the
+ * heap is full prior to the insertion, then the resulting heap will
+ * consist of the smallest @max elements of the original heap and the
+ * new element; the greatest element will be removed from the heap and
+ * returned. Note that the returned element will be the new element
+ * (i.e. no change to the heap) if the new element is greater than all
+ * elements currently in the heap.
```

```
+ */
+extern void *heap_insert(struct ptr_heap *heap, void *p);
+
+
+
+#endif /* _LINUX_PRIO_HEAP_H */
Index: container-2.6.23-mm1/kernel/cpuset.c
===================================================================
--- container-2.6.23-mm1.orig/kernel/cpuset.c
+++ container-2.6.23-mm1/kernel/cpuset.c
@@ -38,6 +38,7 @@
 #include <linux/mount.h>
 #include <linux/namei.h>
 #include <linux/pagemap.h>
+#include <linux/prio_heap.h>
 #include <linux/proc_fs.h>
 #include <linux/rcupdate.h>
 #include <linux/sched.h>
@@ -684,6 +685,36 @@ done:
  /* Don't kfree(doms) -- partition_sched_domains() does that. */
 }

+static int inline started_after_time(struct task_struct *t1,
+         struct timespec *time,
+         struct task_struct *t2)
+{
+ int start_diff = timespec_compare(&t1->start_time, time);
+ if (start_diff > 0) {
+  return 1;
+ } else if (start_diff < 0) {
+  return 0;
+ } else {
+  /*
+   * Arbitrarily, if two processes started at the same
+   * time, we'll say that the lower pointer value
+   * started first. Note that t2 may have exited by now
+   * so this may not be a valid pointer any longer, but
+   * that's fine - it still serves to distinguish
+   * between two tasks started (effectively)
+   * simultaneously.
+   */
+  return t1 > t2;
+ }
+}
+
+static int inline started_after(void *p1, void *p2)
+{
+ struct task_struct *t1 = p1;
```

```
+ struct task_struct *t2 = p2;
+ return started_after_time(t1, &t2->start_time, t2);
+}
+
 /*
  * Call with manage_mutex held.  May take callback_mutex during call.
  */
@@ -691,8 +722,15 @@ done:
 static int update_cpumask(struct cpuset *cs, char *buf)
 {
  struct cpuset trialcs;
- int retval;
- int cpus_changed, is_load_balanced;
+ int retval, i;
+ int is_load_balanced;
+ struct cgroup_iter it;
+ struct cgroup *cgrp = cs->css.cgroup;
+ struct task_struct *p, *dropped;
+ /* Never dereference latest_task, since it's not refcounted */
+ struct task_struct *latest_task = NULL;
+ struct ptr_heap heap;
+ struct timespec latest_time = { 0, 0 };

  /* top_cpuset.cpus_allowed tracks cpu_online_map; it's read-only */
  if (cs == &top_cpuset)
@@ -719,14 +757,73 @@ static int update_cpumask(struct cpuset
  if (retval < 0)
   return retval;

- cpus_changed = !cpus_equal(cs->cpus_allowed, trialcs.cpus_allowed);
+ /* Nothing to do if the cpus didn't change */
+ if (cpus_equal(cs->cpus_allowed, trialcs.cpus_allowed))
+  return 0;
+ retval = heap_init(&heap, PAGE_SIZE, GFP_KERNEL, &started_after);
+ if (retval)
+  return retval;
+
  is_load_balanced = is_sched_load_balance(&trialcs);

  mutex_lock(&callback_mutex);
  cs->cpus_allowed = trialcs.cpus_allowed;
  mutex_unlock(&callback_mutex);

- if (cpus_changed && is_load_balanced)
+ again:
+ /*
+  * Scan tasks in the cpuset, and update the cpumasks of any
+  * that need an update. Since we can't call set_cpus_allowed()
```

```
+  * while holding tasklist_lock, gather tasks to be processed
+  * in a heap structure. If the statically-sized heap fills up,
+  * overflow tasks that started later, and in future iterations
+  * only consider tasks that started after the latest task in
+  * the previous pass. This guarantees forward progress and
+  * that we don't miss any tasks
+  */
+ heap.size = 0;
+ cgroup_iter_start(cgrp, &it);
+ while ((p = cgroup_iter_next(cgrp, &it))) {
+  /* Only affect tasks that don't have the right cpus_allowed */
+  if (cpus_equal(p->cpus_allowed, cs->cpus_allowed))
+   continue;
+  /*
+   * Only process tasks that started after the last task
+   * we processed
+   */
+  if (!started_after_time(p, &latest_time, latest_task))
+   continue;
+  dropped = heap_insert(&heap, p);
+  if (dropped == NULL) {
+   get_task_struct(p);
+  } else if (dropped != p) {
+   get_task_struct(p);
+   put_task_struct(dropped);
+  }
+ }
+ cgroup_iter_end(cgrp, &it);
+ if (heap.size) {
+  for (i = 0; i < heap.size; i++) {
+   struct task_struct *p = heap.ptrs[i];
+   if (i == 0) {
+    latest_time = p->start_time;
+    latest_task = p;
+   }
+   set_cpus_allowed(p, cs->cpus_allowed);
+   put_task_struct(p);
+  }
+  /*
+   * If we had to process any tasks at all, scan again
+   * in case some of them were in the middle of forking
+   * children that didn't notice the new cpumask
+   * restriction.  Not the most efficient way to do it,
+   * but it avoids having to take callback_mutex in the
+   * fork path
+   */
+  goto again;
+ }
```

```
+ heap_free(&heap);
+ if (is_load_balanced)
   rebuild_sched_domains();

  return 0;
Index: container-2.6.23-mm1/kernel/sched.c
===================================================================
--- container-2.6.23-mm1.orig/kernel/sched.c
+++ container-2.6.23-mm1/kernel/sched.c
@@ -4366,8 +4366,21 @@ long sched_setaffinity(pid_t pid, cpumas

  cpus_allowed = cpuset_cpus_allowed(p);
  cpus_and(new_mask, new_mask, cpus_allowed);
+ again:
  retval = set_cpus_allowed(p, new_mask);

+ if (!retval) {
+  cpus_allowed = cpuset_cpus_allowed(p);
+  if (!cpus_subset(new_mask, cpus_allowed)) {
+   /*
+    * We must have raced with a concurrent cpuset
+    * update. Just reset the cpus_allowed to the
+    * cpuset's cpus_allowed
+    */
+   new_mask = cpus_allowed;
+   goto again;
+  }
+ }
 out_unlock:
  put_task_struct(p);
  mutex_unlock(&sched_hotcpu_mutex);
Index: container-2.6.23-mm1/lib/prio_heap.c
===================================================================
--- /dev/null
+++ container-2.6.23-mm1/lib/prio_heap.c
@@ -0,0 +1,70 @@
+/*
+ * Simple insertion-only static-sized priority heap containing
+ * pointers, based on CLR, chapter 7
+ */
+
+#include <linux/slab.h>
+#include <linux/prio_heap.h>
+
+int heap_init(struct ptr_heap *heap, size_t size, gfp_t gfp_mask,
+       int (*gt)(void *, void *))
+{
+ heap->ptrs = kmalloc(size, gfp_mask);
```

```
+ if (!heap->ptrs)
+  return -ENOMEM;
+ heap->size = 0;
+ heap->max = size / sizeof(void *);
+ heap->gt = gt;
+ return 0;
+}
+
+void heap_free(struct ptr_heap *heap)
+{
+ kfree(heap->ptrs);
+}
+
+void *heap_insert(struct ptr_heap *heap, void *p)
+{
+ void *res;
+ void **ptrs = heap->ptrs;
+ int pos;
+
+ if (heap->size < heap->max) {
+  /* Heap insertion */
+  int pos = heap->size++;
+  while (pos > 0 && heap->gt(p, ptrs[(pos-1)/2])) {
+   ptrs[pos] = ptrs[(pos-1)/2];
+   pos = (pos-1)/2;
+  }
+  ptrs[pos] = p;
+  return NULL;
+ }
+
+ /* The heap is full, so something will have to be dropped */
+
+ /* If the new pointer is greater than the current max, drop it */
+ if (heap->gt(p, ptrs[0]))
+  return p;
+
+ /* Replace the current max and heapify */
+ res = ptrs[0];
+ ptrs[0] = p;
+ pos = 0;
+
+ while (1) {
+  int left = 2 * pos + 1;
+  int right = 2 * pos + 2;
+  int largest = pos;
+  if (left < heap->size && heap->gt(ptrs[left], p))
+   largest = left;
+  if (right < heap->size && heap->gt(ptrs[right], ptrs[largest]))
```

```
+   largest = right;
+  if (largest == pos)
+   break;
+  /* Push p down the heap one level and bump one up */
+  ptrs[pos] = ptrs[largest];
+  ptrs[largest] = p;
+  pos = largest;
+ }
+ return res;
+}
Index: container-2.6.23-mm1/lib/Makefile
===================================================================
--- container-2.6.23-mm1.orig/lib/Makefile
+++ container-2.6.23-mm1/lib/Makefile
@@ -6,7 +6,7 @@ lib-y := ctype.o string.o vsprintf.o cmd
   rbtree.o radix-tree.o dump_stack.o \
   idr.o int_sqrt.o bitmap.o extable.o prio_tree.o \
   sha1.o irq_regs.o reciprocal_div.o argv_split.o \
-  proportions.o
+  proportions.o prio_heap.o

 lib-$(CONFIG_MMU) += ioremap.o pagewalk.o
 lib-$(CONFIG_SMP) += cpumask.o
```

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

---

Subject: Re: [RFC] cpuset update_cgroup_cpus_allowed
Posted by David Rientjes on Tue, 16 Oct 2007 18:27:15 GMT
View Forum Message <> Reply to Message

On Tue, 16 Oct 2007, Paul Jackson wrote:

> David wrote:
> > Why can't you just add a helper function to sched.c:
> >
> > void set_hotcpus_allowed(struct task_struct *task,
> >   cpumask_t cpumask)
> > {
> >   mutex_lock(&sched_hotcpu_mutex);
> >   set_cpus_allowed(task, cpumask);
> >   mutex_unlock(&sched_hotcpu_mutex);
> > }
> >
> > And then change each task's cpus_allowed via that function instead of
> > set_cpus_allowed() directly?

>
> I guess this would avoid race conditions within the set_cpus_allowed()
> routine, between its code to read the cpu_online_map and set the tasks
> cpus_allowed ... though if that's useful, don't we really need to add
> locking/unlocking on sched_hotcpu_mutex right inside the
> set_cpus_allowed() routine, for all users of set_cpus_allowed ??
>

Not necessarily because migration only occurs to any online cpu in the
mask, it won't attempt to migrate it to some cpu that has been downed.

> > My solution may be worse than that. Because set_cpus_allowed() will
> > fail if asked to set a non-overlapping cpumask, my solution could never
> > terminate. If asked to set a cpusets cpus to something that went off
> > line right then, this I'd guess this code could keep looping forever,
> > looking for cpumasks that didn't match, and then not noticing that it
> > was failing to set them so as they would match.
>
> These races involve reading the tasks cpuset cpus_allowed mask, reading
> the online map, and both reading and writing the tasks task_struct
> cpus_allowed. Unless one holds the relevant lock for the entire
> interval surrounding the critical accesses to these values, it won't do
> any good that I can see. Just briefly holding a lock around each
> separate access is useless.
>

It's not useless since a cpu hot-unplug event automatically updates the
cpus_allowed for all cpusets in the hierarchy by removing it.

You could protect the entire reassignment with the same mutex since it's
so lightly contended, but it's unnecessary. All it would guarantee is
that cpuset->cpus_allowed is consistent with task->cpus_allowed for all
tasks in cpuset.

Something like the following (untested) patch, but adding a
set_hotcpus_allowed() helper would also work if it's protected by
sched_hotcpu_mutex.
---
 include/linux/sched.h | 8 +++++
 kernel/cpuset.c | 79 ++++++++++++++++++++++++++++++----------
 kernel/sched.c | 13 ++++++--
 mm/pdflush.c | 3 --
 4 files changed, 64 insertions(+), 39 deletions(-)

diff --git a/include/linux/sched.h b/include/linux/sched.h
--- a/include/linux/sched.h
+++ b/include/linux/sched.h
@@ -1391,6 +1391,8 @@ static inline void put_task_struct(struct task_struct *t)

```
 #ifdef CONFIG_SMP
 extern int set_cpus_allowed(struct task_struct *p, cpumask_t new_mask);
+extern void sched_hotcpu_lock(void);
+extern void sched_hotcpu_unlock(void);
 #else
 static inline int set_cpus_allowed(struct task_struct *p, cpumask_t new_mask)
 {
@@ -1398,6 +1400,12 @@ static inline int set_cpus_allowed(struct task_struct *p, cpumask_t
new_mask)
   return -EINVAL;
  return 0;
 }
+static inline void sched_hotcpu_lock(void)
+{
+}
+static inline void sched_hotcpu_unlock(void)
+{
+}
 #endif

 extern unsigned long long sched_clock(void);
diff --git a/kernel/cpuset.c b/kernel/cpuset.c
--- a/kernel/cpuset.c
+++ b/kernel/cpuset.c
@@ -706,40 +706,51 @@ done:
  */
 static void update_cgroup_cpus_allowed(struct cgroup *cont, cpumask_t *cpus)
 {
- int need_repeat = true;
-
- while (need_repeat) {
-  struct cgroup_iter it;
-  const int ntasks = 10;
-  struct task_struct *tasks[ntasks];
-  struct task_struct **p, **q;
-
-  need_repeat = false;
-  p = tasks;
-
-  cgroup_iter_start(cont, &it);
-  while (1) {
-   struct task_struct *t;
-
-   t = cgroup_iter_next(cont, &it);
-   if (!t)
-    break;
-   if (cpus_equal(*cpus, t->cpus_allowed))
```

```
-    continue;
-   if (p == tasks + ntasks) {
-    need_repeat = true;
-    break;
-   }
-   get_task_struct(t);
-   *p++ = t;
-  }
-  cgroup_iter_end(cont, &it);
+ struct cgroup_iter it;
+ struct task_struct *p, **tasks;
+ const int max_tasks = 10;
+ int nr_tasks;
+ int i = 0;
+
+ cgroup_iter_start(cont, &it);
+repeat:
+ for (nr_tasks = max_tasks - 1;
+      (p = cgroup_iter_next(cont, &it)); nr_tasks--) {
+  /*
+   * If the cpumask is already equal for this task, there's no
+   * reason to call set_cpus_allowed() because no change is
+   * needed and no migration is required.
+   */
+  if (cpus_equal(p->cpus_allowed, *cpus))
+   continue;

-  for (q = tasks; q < p; q++) {
-   set_cpus_allowed(*q, *cpus);
-   put_task_struct(*q);
-  }
+  /*
+   * Save a reference to the task structure so it doesn't exit
+   * prematurely.
+   */
+  get_task_struct(p);
+  tasks[i++] = p;
+  if (!nr_tasks)
+   goto set_allowed;
+ }
+ cgroup_iter_end(cont, &it);
+
+set_allowed:
+ while (--i >= 0) {
+  /*
+   * Update the cpus_allowed for this task and migrate it if
+   * necessary.  Then, decrement the task's usage counter.
+   */
```

```
+  set_cpus_allowed(tasks[i], *cpus);
+  put_task_struct(tasks[i]);
  }
+
+ /*
+  * We may need to continue iterating through more tasks if we exhausted
+  * our stack from the previous set.
+  */
+ if (!nr_tasks)
+  goto repeat;
 }

 /*
@@ -779,11 +790,13 @@ static int update_cpumask(struct cpuset *cs, char *buf)
  if (cpus_equal(cs->cpus_allowed, trialcs.cpus_allowed))
   return 0;

+ sched_hotcpu_lock();
  mutex_lock(&callback_mutex);
  cs->cpus_allowed = trialcs.cpus_allowed;
  mutex_unlock(&callback_mutex);
-
  update_cgroup_cpus_allowed(cs->css.cgroup, &cs->cpus_allowed);
+ sched_hotcpu_unlock();
+
  rebuild_sched_domains();
  return 0;
 }
diff --git a/kernel/sched.c b/kernel/sched.c
--- a/kernel/sched.c
+++ b/kernel/sched.c
@@ -51,7 +51,6 @@
 #include <linux/rcupdate.h>
 #include <linux/cpu.h>
 #include <linux/cpuset.h>
-#include <linux/cgroup.h>
 #include <linux/percpu.h>
 #include <linux/cpu_acct.h>
 #include <linux/kthread.h>
@@ -363,6 +362,16 @@ struct rq {
 static DEFINE_PER_CPU_SHARED_ALIGNED(struct rq, runqueues);
 static DEFINE_MUTEX(sched_hotcpu_mutex);

+void sched_hotcpu_lock(void)
+{
+ mutex_lock(&sched_hotcpu_mutex);
+}
+
```

```
+void sched_hotcpu_unlock(void)
+{
+ mutex_unlock(&sched_hotcpu_mutex);
+}
+
 static inline void check_preempt_curr(struct rq *rq, struct task_struct *p)
 {
  rq->curr->sched_class->check_preempt_curr(rq, p);
@@ -4365,11 +4374,9 @@ long sched_setaffinity(pid_t pid, cpumask_t new_mask)
  if (retval)
   goto out_unlock;

- cgroup_lock();
  cpus_allowed = cpuset_cpus_allowed(p);
  cpus_and(new_mask, new_mask, cpus_allowed);
  retval = set_cpus_allowed(p, new_mask);
- cgroup_unlock();

 out_unlock:
  put_task_struct(p);
diff --git a/mm/pdflush.c b/mm/pdflush.c
--- a/mm/pdflush.c
+++ b/mm/pdflush.c
@@ -21,7 +21,6 @@
 #include <linux/writeback.h> // Prototypes pdflush_operation()
 #include <linux/kthread.h>
 #include <linux/cpuset.h>
-#include <linux/cgroup.h>
 #include <linux/freezer.h>


@@ -188,10 +187,8 @@ static int pdflush(void *dummy)
   * This is needed as pdflush's are dynamically created and destroyed.
   * The boottime pdflush's are easily placed w/o these 2 lines.
   */
- cgroup_lock();
  cpus_allowed = cpuset_cpus_allowed(current);
  set_cpus_allowed(current, cpus_allowed);
- cgroup_unlock();

  return __pdflush(&my_work);
 }
```

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

Subject: Re: [RFC] cpuset update_cgroup_cpus_allowed
Posted by Paul Jackson on Tue, 16 Oct 2007 23:14:45 GMT
View Forum Message <> Reply to Message

David wrote:
> Not necessarily because migration only occurs to any online cpu in the
> mask, it won't attempt to migrate it to some cpu that has been downed.
> ...

... one of David or I is insane ... I can't tell which one yet,
perhaps both of us ;).

I'm going to reply to David without all the CC list above.

--
I won't rest till it's the best ...
Programmer, Linux Scalability
Paul Jackson <pj@sgi.com> 1.925.600.0401

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