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