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