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