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
```