Subject: Re: [PATCH 1/1, v6] cgroup/freezer: add per freezer duty ratio control
Posted by Matt Helsley on Thu, 10 Feb 2011 03:04:42 GMT
View Forum Message <> Reply to Message

On Tue, Feb 08, 2011 at 05:05:41PM -0800, jacob.jun.pan@linux.intel.com wrote:
> From: Jacob Pan <jacob.jun.pan@linux.intel.com>
>
> Freezer subsystem is used to manage batch jobs which can start
> stop at the same time. However, sometime it is desirable to let
> the kernel manage the freezer state automatically with a given
> duty ratio.
> For example, if we want to reduce the time that backgroup apps
> are allowed to run we can put them into a freezer subsystem and
> set the kernel to turn them THAWED/FROZEN at given duty ratio.
>
> This patch introduces two file nodes under cgroup
> freezer.duty_ratio_pct and freezer.period_sec
>
> Usage example: set period to be 5 seconds and frozen duty ratio 90%
> [root@localhost aoa]# echo 90 > freezer.duty_ratio_pct
> [root@localhost aoa]# echo 5000 > freezer.period_ms

I kept wondering how this was useful when we've got the "cpu" subsystem because for some reason "duty cycle" made me think this was a scheduling policy knob. In fact, I'm pretty sure it is -- it just happens to sometimes reduce power consumption.

Have you tried using the cpu cgroup subsystem's share to see if it can have a similar effect?

Can you modify the cpu subsystem to enable this instead of putting it into the cgroup freezer subsystem?

The way it oscillates between FROZEN and THAWED also bothers me. The oscillations can be described in millisecond granularity so its possible that reading and manipulating the freezer state from userspace could be largely useless. Also it's not obvious what should happen when the state file is written after the duty cycle has been set (more below).

Perhaps you could fix that up by introducting another state called "DUTY_CYCLE" or something.

What's the overhead of using the freezer as a scheduling mechanism at that granularity? Is it really practical?

What happens to these groups using the duty cycle during suspend and resume? Presumably they won't be accidentally thawed so long as there aren't races between the kernel thread(s) and suspend. I don't think

we've ever had a kernel thread that could thaw a frozen task before
(unless it's part of the resume code itself) so I don't think this race
is covered by existing cgroup freezer code.

Overall I get the feeling this is a scheduling policy knob that doesn't
"belong" in the cgroup freezer subsystem -- though I don't have much
beyond the above questions and my personal aesthetic sense to go on :).

I think Rafael is maintaining the cgroup freezer subsystem since it makes
use of the suspend freezer so I've added him to Cc.

>
> Signed-off-by: Jacob Pan <jacob.jun.pan@linux.intel.com>
> ---
> Documentation/cgroups/freezer-subsystem.txt | 23 +++++
> kernel/cgroup_freezer.c | 132 ++++++++++++++++++++++++++-
> 2 files changed, 154 insertions(+), 1 deletions(-)
>
> diff --git a/Documentation/cgroups/freezer-subsystem.txt
b/Documentation/cgroups/freezer-subsystem.txt
> index 41f37fe..7f06f05 100644
> --- a/Documentation/cgroups/freezer-subsystem.txt
> +++ b/Documentation/cgroups/freezer-subsystem.txt
> @@ -100,3 +100,26 @@ things happens:
>   and returns EINVAL)
> 3) The tasks that blocked the cgroup from entering the "FROZEN"
>   state disappear from the cgroup's set of tasks.
> +
> +In embedded systems, it is desirable to manage group of applications
> +for power saving. E.g. tasks that are not in the foreground may be
> +frozen unfrozen periodically to save power without affecting user

nit: probably should be "frozen and unfrozen periodically"

> +experience. In this case, user/management software can attach tasks
> +into freezer cgroup then specify duty ratio and period that the
> +managed tasks are allowed to run.

And presumably the applications either don't care about their power
consumption, have a bug, or are "malicious" apps -- either way assuming
cooperation from the applications and knowledgable users isn't acceptable.

> +
> +Usage example:
> +Assuming freezer cgroup is already mounted, application being managed
> +are included the "tasks" file node of the given freezer cgroup.
> +To make the tasks frozen at 90% of the time every 5 seconds, do:
> +

```
> +[root@localhost ]# echo 90 > freezer.duty_ratio_pct
> +[root@localhost ]# echo 5000 > freezer.period_ms
> +
> +After that, the application in this freezer cgroup will only be
> +allowed to run at the following pattern.
> +     __                __                  __
> +    |  |<-- 90% frozen -->|  |               |  |
> +____|  |__________________|  |_____________________|  |_____
> +
> +    |<---- 5 seconds ---->|
> diff --git a/kernel/cgroup_freezer.c b/kernel/cgroup_freezer.c
> index e7bebb7..5808f28 100644
> --- a/kernel/cgroup_freezer.c
> +++ b/kernel/cgroup_freezer.c
> @@ -21,6 +21,7 @@
>  #include <linux/uaccess.h>
>  #include <linux/freezer.h>
>  #include <linux/seq_file.h>
> +#include <linux/kthread.h>
>
>  enum freezer_state {
>   CGROUP_THAWED = 0,
> @@ -28,12 +29,28 @@ enum freezer_state {
>   CGROUP_FROZEN,
>  };
>
> +enum duty_ratio_params {
> + FREEZER_DUTY_RATIO = 0,
> + FREEZER_PERIOD,
> +};
> +
> +struct freezer_duty {
> + u32 ratio; /* percentage of time frozen */
> + u32 period_pct_ms; /* one percent of the period in miliseconds */
> +};
> +
>  struct freezer {
>   struct cgroup_subsys_state css;
>   enum freezer_state state;
> + struct freezer_duty duty;
> + struct task_struct *fkh;
>   spinlock_t lock; /* protects _writes_ to state */
>  };
>
> +static struct task_struct *freezer_task;
> +static int try_to_freeze_cgroup(struct cgroup *cgroup, struct freezer *freezer);
> +static void unfreeze_cgroup(struct cgroup *cgroup, struct freezer *freezer);
> +
```

> static inline struct freezer *cgroup_freezer(
>   struct cgroup *cgroup)
> {
> @@ -63,6 +80,31 @@ int cgroup_freezing_or_frozen(struct task_struct *task)
>   return result;
> }
>
> +static DECLARE_WAIT_QUEUE_HEAD(freezer_wait);
> +
> +static int freezer_kh(void *data)

nit: What's "kh"? "Kernel Handler"?

> +{
> + struct cgroup *cgroup = (struct cgroup *)data;
> + struct freezer *freezer = cgroup_freezer(cgroup);
> +
> + do {
> + if (freezer->duty.ratio < 100 && freezer->duty.ratio > 0 &&
> +   freezer->duty.period_pct_ms) {
> +   if (try_to_freeze_cgroup(cgroup, freezer))
> +   pr_info("cannot freeze\n");
> +   msleep(freezer->duty.period_pct_ms *
> +   freezer->duty.ratio);
> +   unfreeze_cgroup(cgroup, freezer);
> +   msleep(freezer->duty.period_pct_ms *
> +   (100 - freezer->duty.ratio));
> + } else {
> +   sleep_on(&freezer_wait);
> +   pr_debug("freezer thread wake up\n");
> + }
> + } while (!kthread_should_stop());
> + return 0;
> +}

Seems to me you could avoid the thread-per-cgroup overhead and the
sleep-loop code by using one timer-per-cgroup. When the timer expires
you freeze/thaw the cgroup associated with the timer, setup the next
wakeup timer, and use only one kernel thread to do it all. If you
use workqueues you might even avoid the single kernel thread.

Seems to me like that'd be a good fit for embedded devices.

> +
> /*
>   * cgroups_write_string() limits the size of freezer state strings to
>   * CGROUP_LOCAL_BUFFER_SIZE
> @@ -150,7 +192,12 @@ static struct cgroup_subsys_state *freezer_create(struct

```
cgroup_subsys *ss,
>  static void freezer_destroy(struct cgroup_subsys *ss,
>         struct cgroup *cgroup)
>  {
> - kfree(cgroup_freezer(cgroup));
> + struct freezer *freezer;
> +
> + freezer = cgroup_freezer(cgroup);
> + if (freezer->fkh)
> +  kthread_stop(freezer->fkh);
> + kfree(freezer);
>  }
>
>  /*
> @@ -282,6 +329,16 @@ static int freezer_read(struct cgroup *cgroup, struct cftype *cft,
>   return 0;
>  }
>
> +static u64 freezer_read_duty_ratio(struct cgroup *cgroup, struct cftype *cft)
> +{
> + return cgroup_freezer(cgroup)->duty.ratio;
> +}
> +
> +static u64 freezer_read_period(struct cgroup *cgroup, struct cftype *cft)
> +{
> + return cgroup_freezer(cgroup)->duty.period_pct_ms * 100;
> +}
> +
>  static int try_to_freeze_cgroup(struct cgroup *cgroup, struct freezer *freezer)
>  {
>   struct cgroup_iter it;
> @@ -368,12 +425,85 @@ static int freezer_write(struct cgroup *cgroup,
>   return retval;
>  }
>
> +#define FREEZER_KH_PREFIX  "freezer_"
> +static int freezer_write_param(struct cgroup *cgroup, struct cftype *cft,
> +  u64 val)
> +{
> + struct freezer *freezer;
> + char thread_name[32];
> + int ret = 0;
> +
> + freezer = cgroup_freezer(cgroup);
> +
> + if (!cgroup_lock_live_group(cgroup))
> +  return -ENODEV;
> +
```

```
> + switch (cft->private) {
> + case FREEZER_DUTY_RATIO:
> +  if (val >= 100 || val < 0) {
> +   ret = -EINVAL;
> +   goto exit;
> +  }
> +  freezer->duty.ratio = val;
```

Why can't val == 100? At that point it's always THAWED and no kernel thread is necessary (just like at 0 it's always FROZEN and no kernel thread is necessary).

```
> +  break;
> + case FREEZER_PERIOD:
> +  if (val)
> +   do_div(val, 100);
> +   freezer->duty.period_pct_ms = val;
```

Wrong indent level at least. Possible bug?
Shouldn't you disallow duty.period_pct_ms being set to 0? Then userspace can pin a kernel thread at 100% cpu just doing freeze/thaws couldn't it?

```
> +  break;
> + default:
> +  BUG();
> + }
> +
> + /* start/stop management kthread as needed, the rule is that
> +  * if both duty ratio and period values are zero, then no management
> +  * kthread is created. when both are non-zero, we create a kthread
> +  * for the cgroup. When user set zero to duty ratio and period again
> +  * the kthread is stopped.
> +  */
> + if (freezer->duty.ratio && freezer->duty.period_pct_ms) {
> +  if (!freezer->fkh) {
> +   snprintf(thread_name, 32, "%s%s", FREEZER_KH_PREFIX,
> +    cgroup->dentry->d_name.name);
> +   freezer->fkh = kthread_run(freezer_kh, (void *)cgroup,
> +     thread_name);
> +   if (IS_ERR(freezer_task)) {
> +    pr_err("create %s failed\n", thread_name);
> +    ret = PTR_ERR(freezer_task);
> +    goto exit;
> +   }
> +  } else
> +   wake_up(&freezer_wait);
> + } else if ((!freezer->duty.ratio || !freezer->duty.period_pct_ms) &&
```

> + freezer->fkh) {
> +   kthread_stop(freezer->fkh);
> +   freezer->fkh = NULL;
> + }
> +
> +exit:
> + cgroup_unlock();
> + return ret;
> +}
> +
> static struct cftype files[] = {
>   {
>   .name = "state",
>   .read_seq_string = freezer_read,
>   .write_string = freezer_write,

It's not clear what should happen when userspace writes the state
file after writing a duty_ratio_pct.

If the new state file write takes priority then:
 Writing THAWED to the state should set duty_ratio_pct to 100.
 Writing FROZEN to the state should set it to 0.

 This means existing code will get the behavior it expects.

Else, if you want duty_ratio_pct to take priority then you ought to make
the state file read-only when duty_ratio_pct is set. Otherwise existing
userspace code will happily chug along without noticing that their groups
aren't doing what they expected. This is also another good reason to
introduce a new state as suggested above (with the tenative name
"DUTY_CYCLE").

>   },
> + {
> + .name = "duty_ratio_pct",
> + .private = FREEZER_DUTY_RATIO,
> + .read_u64 = freezer_read_duty_ratio,
> + .write_u64 = freezer_write_param,
> + },

nit: Why use a u64 for a value that can only be 0-100? (or perhaps
0-1000 if you wanted sub-1% granularity...)

Cheers,
 -Matt Helsley

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