Subject: Re: Re: Hang with fair cgroup scheduler (reproducer is attached.)
Posted by Dmitry Adamushko on Sat, 15 Dec 2007 23:44:55 GMT
View Forum Message <> Reply to Message

On 15/12/2007, Dmitry Adamushko <dmitry.adamushko@gmail.com> wrote:
>
> My analysis was flawed (hmm... me was under control of Belgium beer :-)
>

ok, I've got another one (just in case... well, this late hour to be
blamed now :-/)

according to Dhaval, we have a crash on ia64 (it's also the arch for
the original report) and it's not reproducible on an otherwise similar
(wrt. # of cpus) x86.

(1) The difference that comes first in mind is that ia64 makes use of
__ARCH_WANT_UNLOCKED_CTXSW

dimm@earth:~/storage/kernel/linux-2.6$ grep -rn
__ARCH_WANT_UNLOCKED_CTXSW include/
include/linux/sched.h:947:#ifdef __ARCH_WANT_UNLOCKED_CTXSW
include/asm-mips/system.h:216:#define __ARCH_WANT_UNLOCKED_CTXSW
include/asm-ia64/system.h:259:#define __ARCH_WANT_UNLOCKED_CTXSW

(2) now, in this case (and for SMP)

task_running() effectively becomes { return p->oncpu; }

(3) consider a case of the context switch between prev --> next on CPU #0

'next' has preempted 'prev'

(4) context_swicth() :

next->oncpu becomes '1' as the result of:

[1] context_switch() --> prepare_task_switch() --> prepare_lock_switch(next) -->
next->oncpu = 1

prev->oncpu becomes '0' as the result of:

[2] context_switch() --> finish_task_switch() -->
finish_lock_switch(prev) --> prev->oncpu = 0

[1] takes place at the very _beginning_ of context_switch() _and_ one more thing is that rq->lock gets unlocked.

[2] takes place at the very _end_ of context_switch()

Now recall what's task_running() in our case (it's "return task->oncpu")

As a result, between [1] and [2] we have 2 tasks on a single CPU for which task_running() will return '1' and their runqueue is _unlocked_.

(5) now consider sched_move_task() running on another CPU #1.

due to 'UNLOCKED_CTXSW' it can successfully lock the rq of CPU #0

let's say it's called for 'prev' task (the one being scheduled out on CPU #0 at this very moment)

as we remember, task_running() returns '1' for it (CPU #0 haven't reached yet point [2] as described in (4) above)

'prev' is currently on the runqueue (prev->se.on_rq == 1) and within the tree.

what happens is as follows:

- dequeue_task() removes it from the tree ;
- put_prev_task() makes cfs_rq->curr = NULL ;

se == prev.se here... so e.g. __enqueue_entity() is not called for 'prev'

- set_curr_task() --> set_curr_task_fair()

and here things become interesting.

```
static void set_curr_task_fair(struct rq *rq)
{
	struct sched_entity *se = &rq->curr->se;

	for_each_sched_entity(se)
		set_next_entity(cfs_rq_of(se), se);
}
```

so 'se' actually belongs to the 'next' on CPU #0

next->on_rq == 1 (obviously, as dequeue_task() in sched_move_task() was done for 'prev' !)

and now, set_next_entity() does __dequeue_entity() for 'next' which is
not within the tree !!!
(it's the real 'current' on CPU #0)

that's why the reported oops:

> [<a0000001002e0480>] rb_erase+0x300/0x7e0
> [<a000000100076290>] __dequeue_entity+0x70/0xa0
> [<a000000100076300>] set_next_entity+0x40/0xa0
> [<a0000001000763a0>] set_curr_task_fair+0x40/0xa0
> [<a000000100078d90>] sched_move_task+0x2d0/0x340
> [<a000000100078e20>] cpu_cgroup_attach+0x20/0x40

or maybe there is also a possibility of the rb-tree being corrupted as
a result and having a crash somewhere later (the original report had
another backtrace)

hum... does this analysis make sense to somebody else now?

--
Best regards,
Dmitry Adamushko