
Subject: Re: [PATCH] ia64 sn xpc: Convert to use kthread API.

Posted by [ebiederm](#) on Fri, 27 Apr 2007 20:33:32 GMT

[View Forum Message](#) <> [Reply to Message](#)

Dean Nelson <dcn@sgi.com> writes:

> On Fri, Apr 27, 2007 at 12:34:02PM -0600, Eric W. Biederman wrote:

>> Dean Nelson <dcn@sgi.com> writes:

>>>

>>> Christoph is correct in that XPC has a single thread that exists throughout
>>> its lifetime, another set of threads that exist for the time that active
>>> contact with other XPCs running on other SGI system partitions exists, and
>>> finally there is a pool of threads that exist on an as needed basis once
>>> a channel connection has been established between two partitions.

>>>

>>> In principle I approve of the kthread API and its use as opposed to what
>>> XPC currently does (calls `kernel_thread()`, `daemonize()`,
> `wait_for_completion()`,
>>> and `complete()`). So Christoph's patch that changes the single long-lived
>>> thread to use `kthread_stop()` and `kthread_should_stop()` is appreciated.

>>>

>>> But the fact that another thread, started at the `xpc_init()` time, that does
>>> discovery of other SGI system partitions wasn't converted points out a
>>> weakness in either my thinking or the kthread API. This discovery thread
>>> does its job and then exits. Should XPC be `rmmod'd` while the discovery
>>> thread is still running we would need to do a `kthread_stop()` against it.
>>> But `kthread_stop()` isn't set up to deal with a task that has already exited.
>>> And if what once was the task structure of this exited task has been
>>> reallocated to another new task, we'd end up stopping it should it be
>>> operating under the kthread API, or possibly waiting a very long time
>>> for it to exit if it is not.

>>>

>>> Patches are currently under development to allow kthreads to exit
>>> before `kthread_stop` is called. The big thing is that once we allow
>>> kernel threads that exited by themselves to be reaped by `kthread_stop`
>>> we have some significant work to do.

>>>

>>> I'm also a little uneasy that `kthread_stop()` has an "only one thread can
>>> stop another thread at a time" design. It's a potential bottleneck on
>>> very large systems where threads are blocked and unable to respond to a
>>> `kthread_should_stop()` for some period of time.

>>>

>>> There are already patches out there to fix this issue.

>>>

>>> XPC is in need of threads that can block indefinitely, which is why XPC
>>> is in the business of maintaining a pool of threads. Currently there is
>>> no such capability (that I know of) that is provided by linux. Workqueues
>>> can't block indefinitely.

>>
>> I'm not certain I understand this requirement. Do you mean block indefinitely
>> unless requested to stop?
>
> These threads can block waiting for a hardware DMA engine, which has a 28
> second timeout setpoint.

Ok. So this is an interruptible sleep?
Do you have any problems being woken up out of that interruptible sleep
by kthread_stop?

I am in the process of modifying kthread_stop to wake up thread in an
interruptible sleep and set signal_pending, so they will break out.

>> > And for performance reasons these threads need to be able to be created
>> > quickly. These threads are involved in delivering messages to XPC's users
>> > (like XPNET) and we had latency issues that led us to use kernel_thread()
>> > directly instead of the kthread API. Additionally, XPC may need to have
>> > hundreds of these threads active at any given time.
>>
>> Ugh. Can you tell me a little more about the latency issues?
>
> After placing a message in a local message queue, one SGI system partition
> will interrupt another to retrieve the message. We need to minimize the
> time from entering XPC's interrupt handler to the time that the message
> can be DMA transferred and delivered to the consumer (like XPNET) to
> whom it was sent.
>
>> Is having a non-halting kthread_create enough to fix this?
>> So you don't have to context switch several times to get the
>> thread running?
>>
>> Or do you need more severe latency reductions?
>>
>> The more severe fix would require some significant changes to copy_process
>> and every architecture would need to be touched to fix up copy_thread.
>> It is possible, it is a lot of work, and the reward is far from obvious.
>
> I think a non-halting kthread_create() should be sufficient. It is in
> effect what XPC has now in calling kernel_thread() directly.

A little different but pretty close.

We call kthread_create() it prepares everything and places it on
a queue and wakes up kthreadd.

kthreadd then wakes up and forks the thread.

After the thread has finishing setting up it will call complete on a completion so kthread_create can continue on it's merry way but it should not need to go to sleep waiting for someone to call kthread_bind.

But if you can live with what I have just described that will be easy to code up.

It is a little slower then kernel_thread but hopefully not much.

> Taking it one step further, if you added the notion of a thread pool,
> where upon exit, a thread isn't destroyed but rather is queued ready to
> handle the next kthread_create_quick() request.

That might happen. So far I am avoiding the notion of a thread pool for as long as I can. There is some sense in it, especially in generalizing the svc thread pool code from nfs. But if I don't have to go there I would prefer it.

>> > I think it would be great if the kthread API (or underlying implementation)
>> > could be changed to handle these issues. I'd love for XPC to not have to
>> > maintain this sort of thing itself.

>>

>> Currently daemonize is a serious maintenance problem.

>>

>> Using daemonize and kernel_thread to create kernel threads is a blocker
>> in implementing the pid namespace because of their use of pid_t.

>>

>> So I am motivated to get this fixed.

>

> This would also address the problems we see with huge pid spaces for
> kernel threads on our largest machines. In the example from last week,
> we had 10 threads each on 4096 cpus. If we reworked work_queues to use
> the kthread_create_nonblocking() thread pool, we could probably collapse
> the need for having all of those per-task, per-cpu work queues.

Patches have already been sent (and I don't think found problems with) that make kthreadd pid == 2, and they also modify daemonize to reparent to kthreadd instead of init.

Eric

Containers mailing list

Containers@lists.linux-foundation.org

<https://lists.linux-foundation.org/mailman/listinfo/containers>
