Subject: Re: [PATCH] ia64 sn xpc: Convert to use kthread API.
Posted by Dean Nelson on Fri, 27 Apr 2007 17:41:42 GMT
View Forum Message <> Reply to Message

On Thu, Apr 19, 2007 at 04:51:03PM -0700, Andrew Morton wrote:
> Another driver which should be fully converted to the kthread API:
> kthread_stop() and kthread_should_stop().
>
> And according to my logs, this driver was added to the tree more than
> a year _after_ the kthread interface was made available.
>
> This isn't good.

On Sun, Apr 22, 2007 at 09:36:47PM +0100, Christoph Hellwig wrote:
> On Thu, Apr 19, 2007 at 01:58:44AM -0600, Eric W. Biederman wrote:
> > From: Eric W. Biederman <ebiederm@xmission.com>
> >
> > This patch starts the xpc kernel threads using kthread_run
> > not a combination of kernel_thread and daemonize. Resuling
> > in slightly simpler and more maintainable code.
>
> This driver is a really twisted maze. It has a lot of threads,
> some of them running through the whole lifetime of the driver,
> some short-lived and some in a sort of a pool.
>
> The patch below fixes up the long-lived thread as well as fixing
> gazillions of leaks in the init routine by switching to proper
> goto-based unwinding.

I see that the setting of 'xpc_rsvd_page->vars_pa = 0;' in xpc_init() is
considered a leak by Christoph (hch), but it really is not. If you look at
xpc_rsvd_page_init() where it is set up, you might see that the reserved page
is something XPC gets from SAL who created it at system boot time. If XPC is
rmmod'd and insmod'd again, it will be handed the same page of memory by SAL.
So there is no memory leak here.

As for the other suspected leaks mentioned I'm not sure what they could be.
It may be the fact that XPC continues coming up when presented with error
returns from register_reboot_notifier() and register_die_notifier(). There
is no leak in this, just simply a loss of an early notification to other
SGI system partitions that this partition is going down. A fact they will
sooner or later discover on their own. A notice of this degraded
functionality is written to the console. And the likelyhood that these
functions should ever return an error is very, very small (currently,
neither of them has an error return).

>From my experience the goto-based unwinding of error returns is not necessarily
a superior approach. I spent a month tracking down a difficult to reproduce

problem that ended up being an error return jumping to the wrong label in a goto-based unwind. Problems can arise with either approach. I've also seen the compiler generate less code for the non-goto approach. I'm not a compiler person so I can't explain this, nor can I say that it's always the case, but at one time when I did a bake off between the two approaches the non-goto approach generated less code.

Having said this I have no problem with switching to a goto-based unwinding of errors if that is what the community prefers. I personally find it more readable than the non-goto approach.

> Note that thread pools are something we have in a few places,
> and might be worth handling in the core kthread infrastructure,
> as tearing down pools will get a bit complicated using the
> kthread APIs.

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 weekness 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.

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.

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.

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.

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.

Dean

_____
Containers mailing list
Containers@lists.linux-foundation.org
https://lists.linux-foundation.org/mailman/listinfo/containers