Subject: Re: [RFC] kernel/pid.c pid allocation wierdness
Posted by ebiederm on Fri, 16 Mar 2007 21:18:06 GMT
View Forum Message <> Reply to Message

William Lee Irwin III <wli@holomorphy.com> writes:
>
> On Fri, Mar 16, 2007 at 07:04:28AM -0600, Eric W. Biederman wrote:
>> Grr. s/patricia tree/fib tree/. We use that in the networking for
>> the forwarding information base and I got mis-remembered it. Anyway
>> the interesting thing with the binary version of radix tree is that
>> path compression is well defined. Path compression when you have
>> multi-way branching is much more difficult.
>
> Path compression isn't a big deal for multiway branching. I've usually
> done it by aligning nodes and or'ing the number of levels to skip into
> the lower bits of the pointer.

Hmm. I guess what I have seen it that it was simply more difficult
because there were fewer opportunities the bigger the branching factor
but I haven't looked at it very closely.

> On Fri, Mar 16, 2007 at 07:04:28AM -0600, Eric W. Biederman wrote:
>> Sure. One of the reasons to be careful with switching data
>> structures. Currently the hash tables typically operate at 10:1
>> unsued:used entries. 4096 entries and 100 processes.
>
> That would be 40:1, which is "worse" in some senses. That's not
> going to fly well when pid namespaces proliferate.

Agreed, currently the plan it to add an namespace parameter to hash table
compares during lookups. Allocating hash tables at run time is almost
impossible to do reliably because of the multiple page allocations.

> On Fri, Mar 16, 2007 at 07:04:28AM -0600, Eric W. Biederman wrote:
>> The current work actually focuses on changing the code so we reduce
>> the total number of hash table looks ups, but persistently storing
>> struct pid pointers instead of storing a pid_t. This has a lot
>> of benefits when it comes to implementing a pid namespace but the
>> secondary performance benefit is nice as well.
>
> I can't quite make out what you mean by all that. struct pid is already
> what's in the hashtable.

Yes. But I have given it a life of it's own as well. Which means instead
of caching a pid_t value in a long lived data structure we can hold
a struct pid *. So that means we have fewer total hash table look
ups.

> On Fri, Mar 16, 2007 at 07:04:28AM -0600, Eric W. Biederman wrote:
>> Although my preliminary concern was the increase in typical list
>> traversal length during lookup. The current hash table typically does
>> not have collisions so normally there is nothing to traverse.
>
> Define "normally;" 10000 threads and/or processes can be standard for
> some affairs.

When I did a quick survey of systems I could easily find everything
was much lower than. I wasn't been able to find those setups in my
quick survey. I was looking for systems with long hash chains to
justify a data structure switch especially systems that needed to
push up the default pid limit, but I didn't encounter them.

So that said to me the common case was well handled by the current
setup. Especially where even at 10000 we only have normal hash chain
lengths of 3 to 4 (3 to 5?). I did a little modeling and our hash
function was good enough that it generally gave a good distribution of
pid values across the buckets.

My memory is something like the really nasty cases only occur when
we start pushing /proc/sys/kernel/pid_max above it's default at 32768.

Our worst case has pid hash chains of 1k entries which clearly sucks.

> William Lee Irwin III <wli@holomorphy.com> writes:
>>> RCU'ing radix trees is trendy but the current implementation needs a
>>> spinlock where typically radix trees do not need them for RCU. I'm
>>> talking this over with others interested in lockless radix tree
>>> algorithms for reasons other than concurrency and/or parallelism.
>
> On Fri, Mar 16, 2007 at 07:04:28AM -0600, Eric W. Biederman wrote:
>> Sure. I was thinking of the general class of data structures not the
>> existing kernel one. The multi-way branching and lack of comparisons
>> looks interesting as a hash table replacement. In a lot of cases
>> storing hash values in a radix tree seems a very interesting
>> alternative to a traditional hash table (and in that case the keys
>> are randomly distributed and can easily be made dense).   For pids
>> hashing the values first seems like a waste, and
>
> Comparisons vs. no comparisons is less interesting than cachelines
> touched. B+ would either make RCU inconvenient or want comparisons by
> dereferencing, which raises the number of cachelines touched.

Agreed. Hmm. I didn't say that too well, I was thinking of the lack
of comparisons implying fewer cache line touches.

> William Lee Irwin III <wli@holomorphy.com> writes:

>>> I'd say you already have enough evidence to motivate a change of data
>>> structure.
>
> On Fri, Mar 16, 2007 at 07:04:28AM -0600, Eric W. Biederman wrote:
>> I have enough to know that a better data structure could improve
>> things. Getting something that is clearly better than what
>> we have now is difficult. Plus I have a lot of other patches to
>> coordinate. I don't think the current data structure is going to
>> fall over before I get the pid namespace implemented so that is
>> my current priority.
>
> I'll look into the data structure code, then.

Thanks.

>
> On Fri, Mar 16, 2007 at 07:04:28AM -0600, Eric W. Biederman wrote:
>> My secondary use is that I need something for which I can do a full
>> traversal with for use in readdir. If we don't have a total ordering
>> that is easy and safe to store in the file offset field readdir can
>> loose entries. Currently I use the pid value itself for this.
>> Simply limiting tree height in the case of pids which are relatively
>> dense is likely to be enough.
>
> Manfred's readdir code should've dealt with that at least partially.

Partially sounds correct. I had to replace it recently because it
was possible for readdir to skip a process that existed for the entire
length of an opendir, readdir_loop, closedir session. It took a
process exiting to trigger it so it was rare but the semantics were
impossible for user space to work around.

The problem is if the process we stop on disappears things must be
well ordered enough that we can find the next succeeding process.
The previous code (which I assume Manfred did from skimming the
changelog) would simply count forward in a fixed number of processes
in the process list (when the process it had stop on died). Which
since we always append to the end would of the task list would skip
processes immediately after those that had exited.

> B+ trees also resolve that quite well, despite their other issues.
> Sacrificing fully lockless RCU on B+ trees and wrapping writes with
> spinlocks should allow pid numbers to be stored alongside pointers
> in leaf nodes at the further cost of a branching factor reduction.
> TLB will at least be conserved even with a degraded branching factor.
>
> Anyway, I'm loath to use lib/radix-tree.c but a different radix tree
> implementation I could run with. I've gone over some of the other

> alternatives. Give me an idea of where you want me to go with the data
> structure selection and I can sweep that up for you. I'm not attached
> to any particular one, though I am repelled from one in particular
> (which I can do anyway even if only for comparison purposes, though if
> it happens to be best I'll concede and let it through anyway).
>
> I can also defer the data structure switch to you if you really want
> to reserve that for yourself.

No. If someone else is interested and can do the work I don't want
to reserve it for myself.

As long as I get to help review the changes I don't have any real
preferences for data structures as long it meets the needs of the pid
lookup.

I should mention there is also a subtle issue at the leaf nodes. The
pid namespaces will be hierarchical with parents fully containing
their children. Each struct pid will appear in it's current pid
namespace and all parent pid namespaces (or else we can't use
traditional unix process control functions (like kill, wait, and
ptrace)). Each struct pid will be assigned a different pid_t value in
each pid namespace.

When we want the pid value of a process that isn't current there will
be a function pid_nr() that looks at our current pid namespace and
finds the pid_t value that corresponds to that pid namespace.

In general I don't expect the hierarchy of pid namespaces to be very
deep 1 for the traditional case and when we are actually taking
advantage of the pid namespaces 2 or 3, and so we might be able to
optimize taking that into account as long as the interfaces don't have
that assumption. That observation does mean pid_nr can easily be a
walk through all of the possibilities.

The implication there is that we might end up with:

```
struct pid_lookup_entry {
	pid_t nr;
	struct list_head lookup_list;
	struct pid_namespace *ns;
	struct pid *pid;
	struct list_head pid_list;
};

struct pid
{
 atomic_t count;
	 struct list_head pid_list;
```

```
 /* lists of tasks that use this pid */
 struct hlist_head tasks[PIDTYPE_MAX];
 struct rcu_head rcu;
};
```

Alternatively it might just be:

```
struct pid
{
 atomic_t count;
 /* lists of tasks that use this pid */
 struct hlist_head tasks[PIDTYPE_MAX];
 struct rcu_head rcu;
        struct {
         struct pid_namespace *ns;
                pid_t nr;
        } pids[PID_MAX_DEPTH];
};
```

There are a lot of variations on that theme and it really depends
on the upper level data structures which one we pick.

Eric

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