Subject: Re: [RFC] kernel/pid.c pid allocation wierdness
Posted by William Lee Irwin III on Wed, 14 Mar 2007 15:03:06 GMT
View Forum Message <> Reply to Message

On Wed, Mar 14, 2007 at 08:12:35AM -0600, Eric W. Biederman wrote:
> If we do dig into this more we need to consider a radix_tree to hold
> the pid values. That could replace both the pid map and the hash
> table, gracefully handle but large and small pid counts, might
> be a smidgin simpler, possibly be more space efficient, and it would
> more easily handle multiple pid namespaces. The downside to using a
> radix tree is that is looks like it will have more cache misses for
> the normal pid map size, and it is yet another change that we would
> need to validate.

Radix trees' space behavior is extremely poor in sparsely-populated
index spaces. There is no way they would save space or even come close
to the current space footprint.

Lock contention here would be a severe functional regression (note
"functional," not "performance;" the lock contention surrounding these
affairs takes down systems vs. mere slowdown nonsense), so it would
necessarily depend on lockless radix tree code for correctness.

The comment block describing the hashtable locking is stale and should
have been updated in tandem with the RCU changes.

-- wli

_____
Containers mailing list
Containers@lists.osdl.org
https://lists.osdl.org/mailman/listinfo/containers

Subject: Re: [RFC] kernel/pid.c pid allocation wierdness
Posted by ebiederm on Wed, 14 Mar 2007 16:54:07 GMT
View Forum Message <> Reply to Message

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

> On Wed, Mar 14, 2007 at 08:12:35AM -0600, Eric W. Biederman wrote:
>> If we do dig into this more we need to consider a radix_tree to hold
>> the pid values. That could replace both the pid map and the hash
>> table, gracefully handle but large and small pid counts, might
>> be a smidgin simpler, possibly be more space efficient, and it would
>> more easily handle multiple pid namespaces. The downside to using a
>> radix tree is that is looks like it will have more cache misses for
>> the normal pid map size, and it is yet another change that we would

>> need to validate.
>
> Radix trees' space behavior is extremely poor in sparsely-populated
> index spaces. There is no way they would save space or even come close
> to the current space footprint.

Possibly. We aren't that sparsely populated when it comes to pids.
Hash tables aren't good at saving space either, and when they are space
efficient they are on the edge of long hash chains so they are on
the edge of performance problems. There is at least one variant of the
fib tree that is as space efficient as any binary tree but works by
looking at bits. Not that I think that one makes much sense.

> Lock contention here would be a severe functional regression (note
> "functional," not "performance;" the lock contention surrounding these
> affairs takes down systems vs. mere slowdown nonsense), so it would
> necessarily depend on lockless radix tree code for correctness.

I don't know about the existing in kernel implementations but there is no
reason we could not have an rcu protected radix tree. At which point the
challenges are about the same but we have indexes that would help us find
the next free bit, which could reduce cpu time.

The current pid implementation does not scale to larger process counts
particularly well. The hash table size is fixed so we get a lot of hash
collisions etc. Several other things go wonky as well. The one time
I actually had 64K pids on a system (most of them zombies) it was not
a very pleasant situation.

It isn't common that we push the pid count up, and with the normal pid
counts the current data structures seem very well suited to the
problem. I have been looking at the data structures though in case it
ever changes because the current good behavior seems quite fragile.
Not that I am advocating changing anything yet, but I'm thinking about
it so when we do come to the point where it matters we can make a
reasonable change.

> The comment block describing the hashtable locking is stale and should
> have been updated in tandem with the RCU changes.

Feel free to submit the patch. I didn't make the RCU changes just took
advantage of them.

Eric

Subject: Re: [RFC] kernel/pid.c pid allocation wierdness
Posted by William Lee Irwin III on Thu, 15 Mar 2007 20:26:54 GMT
View Forum Message <> Reply to Message

William Lee Irwin III <wli@holomorphy.com> writes:
>> Radix trees' space behavior is extremely poor in sparsely-populated
>> index spaces. There is no way they would save space or even come close
>> to the current space footprint.

On Wed, Mar 14, 2007 at 10:54:07AM -0600, Eric W. Biederman wrote:
> Possibly. We aren't that sparsely populated when it comes to pids.
> Hash tables aren't good at saving space either, and when they are space
> efficient they are on the edge of long hash chains so they are on
> the edge of performance problems. There is at least one variant of the
> fib tree that is as space efficient as any binary tree but works by
> looking at bits. Not that I think that one makes much sense.

I'd not mind something better than a hashtable. The fib tree may make
more sense than anticipated. It's truly better to switch data structures
completely than fiddle with e.g. hashtable sizes. However, bear in mind
the degenerate space behavior of radix trees in sparse contexts.

William Lee Irwin III <wli@holomorphy.com> writes:
>> Lock contention here would be a severe functional regression (note
>> "functional," not "performance;" the lock contention surrounding these
>> affairs takes down systems vs. mere slowdown nonsense), so it would
>> necessarily depend on lockless radix tree code for correctness.

On Wed, Mar 14, 2007 at 10:54:07AM -0600, Eric W. Biederman wrote:
> I don't know about the existing in kernel implementations but there is no
> reason we could not have an rcu protected radix tree. At which point the
> challenges are about the same but we have indexes that would help us find
> the next free bit, which could reduce cpu time.

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 Wed, Mar 14, 2007 at 10:54:07AM -0600, Eric W. Biederman wrote:
> The current pid implementation does not scale to larger process counts
> particularly well. The hash table size is fixed so we get a lot of hash
> collisions etc. Several other things go wonky as well. The one time
> I actually had 64K pids on a system (most of them zombies) it was not
> a very pleasant situation.

If you've got a microbenchmark that would be convenient for me to use

while addressing it, unless you'd prefer to take it on yourself.
Otherwise I can always write one myself.

On Wed, Mar 14, 2007 at 10:54:07AM -0600, Eric W. Biederman wrote:
> It isn't common that we push the pid count up, and with the normal pid
> counts the current data structures seem very well suited to the
> problem. I have been looking at the data structures though in case it
> ever changes because the current good behavior seems quite fragile.
> Not that I am advocating changing anything yet, but I'm thinking about
> it so when we do come to the point where it matters we can make a
> reasonable change.

I'd say you already have enough evidence to motivate a change of data
structure. Tree hashing (e.g. using balanced search trees in collision
chains) is generally good for eliminating straight-line issues of this
form but the available in-kernel tree structures don't do so well in
concurrent/parallel contexts and the utility of hashing becomes somewhat
questionable afterward given the stringent limits kernel environments
impose on pid spaces. I favor Peter Zijlstra's B+ trees once a few
relatively minor issues are addressed on account of good behavior in
sparse keyspaces, though cleanups (not stylistic) of radix trees' space
behavior may yet render them suitable, and may also be more popular to
pursue.

Basically all that's needed for radix trees' space behavior to get
fixed up is proper path compression as opposed to the ->depth hack.
Done properly it also eliminates the need for a spinlock around the
whole radix tree for RCU.

William Lee Irwin III <wli@holomorphy.com> writes:
>> The comment block describing the hashtable locking is stale and should
>> have been updated in tandem with the RCU changes.

On Wed, Mar 14, 2007 at 10:54:07AM -0600, Eric W. Biederman wrote:
> Feel free to submit the patch. I didn't make the RCU changes just took
> advantage of them.

I needed to note that because it and my description were in direct
conflict. I'd rather leave it for a kernel janitor or someone who needs
to get patches under their belt to sweep up as I've got enough laurels
to rest on and other areas where I can get large amounts of code in
easily enough, provided I get my act together on various fronts.

-- wli

_____

Subject: Re: [RFC] kernel/pid.c pid allocation wierdness
Posted by ebiederm on Fri, 16 Mar 2007 13:04:28 GMT
View Forum Message <> Reply to Message

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

> William Lee Irwin III <wli@holomorphy.com> writes:
>>> Radix trees' space behavior is extremely poor in sparsely-populated
>>> index spaces. There is no way they would save space or even come close
>>> to the current space footprint.
>
> On Wed, Mar 14, 2007 at 10:54:07AM -0600, Eric W. Biederman wrote:
>> Possibly. We aren't that sparsely populated when it comes to pids.
>> Hash tables aren't good at saving space either, and when they are space
>> efficient they are on the edge of long hash chains so they are on
>> the edge of performance problems. There is at least one variant of the
>> fib tree that is as space efficient as any binary tree but works by
>> looking at bits. Not that I think that one makes much sense.
>
> I'd not mind something better than a hashtable. The fib tree may make
> more sense than anticipated. It's truly better to switch data structures
> completely than fiddle with e.g. hashtable sizes. However, bear in mind
> the degenerate space behavior of radix trees in sparse contexts.

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.

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.

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.

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.

Meanwhile an rcu tree would tend towards 2-3 levels which would double
or triple our number of cache line misses on lookup and thus reduce
performance in the normal case.

> William Lee Irwin III <wli@holomorphy.com> writes:
>>> Lock contention here would be a severe functional regression (note
>>> "functional," not "performance;" the lock contention surrounding these
>>> affairs takes down systems vs. mere slowdown nonsense), so it would
>>> necessarily depend on lockless radix tree code for correctness.
>
> On Wed, Mar 14, 2007 at 10:54:07AM -0600, Eric W. Biederman wrote:
>> I don't know about the existing in kernel implementations but there is no
>> reason we could not have an rcu protected radix tree. At which point the
>> challenges are about the same but we have indexes that would help us find
>> the next free bit, which could reduce cpu time.
>
> 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.

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

> On Wed, Mar 14, 2007 at 10:54:07AM -0600, Eric W. Biederman wrote:
>> The current pid implementation does not scale to larger process counts
>> particularly well. The hash table size is fixed so we get a lot of hash
>> collisions etc. Several other things go wonky as well. The one time
>> I actually had 64K pids on a system (most of them zombies) it was not
>> a very pleasant situation.
>
> If you've got a microbenchmark that would be convenient for me to use
> while addressing it, unless you'd prefer to take it on yourself.
> Otherwise I can always write one myself.

I don't. I just have some observations from tracking a bug where we
were creating unreapable zombies, and so the reproducer despite trying
to reap it's zombies created a lot of zombies. I recall that ps was
noticeably slow. Beyond that I don't have good data.

> On Wed, Mar 14, 2007 at 10:54:07AM -0600, Eric W. Biederman wrote:

>> It isn't common that we push the pid count up, and with the normal pid
>> counts the current data structures seem very well suited to the
>> problem. I have been looking at the data structures though in case it
>> ever changes because the current good behavior seems quite fragile.
>> Not that I am advocating changing anything yet, but I'm thinking about
>> it so when we do come to the point where it matters we can make a
>> reasonable change.
>
> I'd say you already have enough evidence to motivate a change of data
> structure.

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.

> Tree hashing (e.g. using balanced search trees in collision
> chains) is generally good for eliminating straight-line issues of this
> form but the available in-kernel tree structures don't do so well in
> concurrent/parallel contexts and the utility of hashing becomes somewhat
> questionable afterward given the stringent limits kernel environments
> impose on pid spaces. I favor Peter Zijlstra's B+ trees once a few
> relatively minor issues are addressed on account of good behavior in
> sparse keyspaces, though cleanups (not stylistic) of radix trees' space
> behavior may yet render them suitable, and may also be more popular to
> pursue.
>
> Basically all that's needed for radix trees' space behavior to get
> fixed up is proper path compression as opposed to the ->depth hack.
> Done properly it also eliminates the need for a spinlock around the
> whole radix tree for RCU.

Yes. Although I have some doubts about path compression. A B+ tree
would be a hair more expensive (as you have to binary search the keys
before descending) and that places a greater emphasis on all of the
keys fitting in one cache line. Still both data structures are
interesting.

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.

> William Lee Irwin III <wli@holomorphy.com> writes:
>>> The comment block describing the hashtable locking is stale and should
>>> have been updated in tandem with the RCU changes.
>
> On Wed, Mar 14, 2007 at 10:54:07AM -0600, Eric W. Biederman wrote:
>> Feel free to submit the patch. I didn't make the RCU changes just took
>> advantage of them.
>
> I needed to note that because it and my description were in direct
> conflict. I'd rather leave it for a kernel janitor or someone who needs
> to get patches under their belt to sweep up as I've got enough laurels
> to rest on and other areas where I can get large amounts of code in
> easily enough, provided I get my act together on various fronts.

Make sense. I still haven't seen that comment yet...
Quite probably because I have a major blind spot for comments
describing how the code works unless they are higher level comments.
I just read the code. I think I only resort to reading the comments
only if I am confused as to what the code is doing or why it is doing
what it is doing.

Eric

Subject: Re: [RFC] kernel/pid.c pid allocation wierdness
Posted by William Lee Irwin III on Fri, 16 Mar 2007 19:46:11 GMT
View Forum Message <> Reply to Message

William Lee Irwin III <wli@holomorphy.com> writes:
>> I'd not mind something better than a hashtable. The fib tree may make
>> more sense than anticipated. It's truly better to switch data structures
>> completely than fiddle with e.g. hashtable sizes. However, bear in mind
>> the degenerate space behavior of radix trees in sparse contexts.

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.

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.

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.

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.

On Fri, Mar 16, 2007 at 07:04:28AM -0600, Eric W. Biederman wrote:
> Meanwhile an rcu tree would tend towards 2-3 levels which would double
> or triple our number of cache line misses on lookup and thus reduce
> performance in the normal case.

It depends on the sort of tree used. For B+ it depends on branching
factors. For hash tries (not previously discussed) with path compression
new levels would only be created when the maximum node size is exceeded.
For radix trees (I'm thinking hand-coded) with path compression it
depends on dispersion especially above the pseudo-levels used for the
lowest bits.

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.

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.

William Lee Irwin III <wli@holomorphy.com> writes:
>> Basically all that's needed for radix trees' space behavior to get
>> fixed up is proper path compression as opposed to the ->depth hack.
>> Done properly it also eliminates the need for a spinlock around the
>> whole radix tree for RCU.

On Fri, Mar 16, 2007 at 07:04:28AM -0600, Eric W. Biederman wrote:
> Yes. Although I have some doubts about path compression. A B+ tree
> would be a hair more expensive (as you have to binary search the keys
> before descending) and that places a greater emphasis on all of the
> keys fitting in one cache line. Still both data structures are
> interesting.

Path compression is no big deal and resolves the radix tree space
issues to my satisfaction. It's actuallly already in lib/radix-tree.c
but in an ineffective form. It's more to do with numbers of cachelines
touched. Having to fetch pid numbers from struct pid's is not going to

be very good on that front.

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.
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.

-- wli

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