This talk is not about XDP: From Resource Limits to SKB Lists

—

David S. Miller, Vancouver, 2018

Two Big Issues

1) Clean and Maintainable Code
2) Isolation...

Clean And Maintainable Code

The story of “ahh, we’ll do that later”

Cleanups put off become increasingly, if not at times exponentially, harder over time.

Look at SKB list handling.

SKB Lists

By-hand list handling, since day one.

Strange state semantics:

- SKB lists are usually circular
- But “not on a list” indicated by a NULL next pointer

The generic list handling exists, has debugging, and should be used... ahhh, we’ll do that later...

But Wait, There's More...

GRO ingress engine has fun semantics:

- Singly linked list
- Special semantics of 'prev' pointer

QDISC transmit layer maintains SKBs on a to_free list

- Meant to keep kfree_skb out of qdisc lock
- Singly linked list

Determine The Scope Of This Mess

Simple recipe:

1) Remove next and prev pointers from sk_buff
2) Run make on net/core/skbuff.o

Boom!

Output from above gives some hints about attack plan

GRO Engine To Lists

Step one was to convert GRO engine to list_head

Already have anonymous struct in sk_buff for rb_tree

Just add list_head next to it

Walk through GRO engine methods adjusting list handling

Many simplifications (f.e. Prev pointer hack was unnecessary)

Wait... GRO Hash Tables?

GRO is weak because all flows sit on a single per-NAPI list

Does not scale with many active parallel flows per cpu

Converting from list_head to a hash table of them is trivial

Followups necessary to adjust limits and book keeping

And fixing bugs... missing skb->next=NULL on overflow

Reassess The Situation

Let's do that again:

1) Remove next and prev pointers from sk_buff
2) Make net/core/skbuff.o
3) Adjust include/linux/skbuff.h based upon build failures
4) ...
5) Get stuck in packet scheduler

Real Attack Plan

Only helpers can access skb->next and skb->prev directly.

Everything else needs to be fixed up.

Core and protocols surprisingly very clean.

Some drivers on the other hand....

Makes list_head conversion involve updating just the relevant helpers

Status

Mostly done.

SCTP and a wireless driver are the main remaining pain points.

Driver is a case of “copy frag list A to frag list B, with alignment and length restrictions A, B and C” Makes for serious spaghetti code and tons of direct accesses to skb->next and skb->prev

SCTP case should be simpler... or maybe not, see next slide.

I have the “flip the switch” patch ready to go once above are done.

Casts to “(struct sk_buff *)”

This is the signature of all of the remaining problems.

Also, casts to “(struct sk_buff_head *)”

If you see code that does this... RUN!

Assumes sk_buff and sk_buff_head both start with next/prev pointers.

After list_head conversion, this is no longer true.

Unfortunately, SCTP has a bunch of this. Working on it!

Resource Sharing Across Network NS

We have many “tables” in the networking code.

Each needs to be at least “keyed” on the network NS

Two models are possible:

1) Global table with NS “key”
2) Segregated tables per-NS

Example Tables

1) ipv4/ipv6 routing tables and caches
2) Socket hashes
3) IP frag queues
4) Neigh/ARP tables
5) Network device lists

Global Table With NS Key

Simplest to implement, and cheapest (wrt. NS creation costs)

Use NS as key and pass to all lookup/insert/delete functions

As a first approximation, this works

But this lacks “object pressure” isolation

Heavy insertion in one NS can hurt other NS instances

Segregated Per-NS Tables

Requires more work and care to implement

NS key is “implicit” so no NS comparisons needed

Per-NS tables means more per-NS base overhead

Limits and sizing need to be reinvestigated carefully because...

Changing per-NS limits can break existing setups

More On Limits

Per-NS limits are hard to select

- Maybe scale based upon some metric?

Global limits satisfy system level constraints

- But allows one NS to hurt others

Maybe we need both?

Global And Per-NS Limits

Purely per-NS limits creates unbounded situation

- With each new NS, globally consumable resources increase

Purely global limits opens a path for abuse

- However, puts a solid cap on system resources

Availability Based Policies

Until system limits are hit, individual NS can use as much as they want.

If system limits are approached, per-NS limits are enforced.

Implies a trimming mechanism of some sort.

Which in turn implies that resources are cached data and can be reconstituted.

Trusted Situations

I know my NS are not going to actively try to grab all system resources.

Ok, you trust your NS but...

Do you trust the network it is on?

Important issue for things like ARP/Neigh caches.

So what do we do?

No single policy satisfies all use cases.

Certain policies can only apply to certain tables (reconstitutability)

Assuming we agree that multiple policies are needed...

... the question becomes one of how to provide this “choice”.

More discussion is definitely needed.