IRQ Suspension: a new, efficient mechanism for packet delivery.

Joe Damato
jdamato@fastly.com
Netdev 0x19

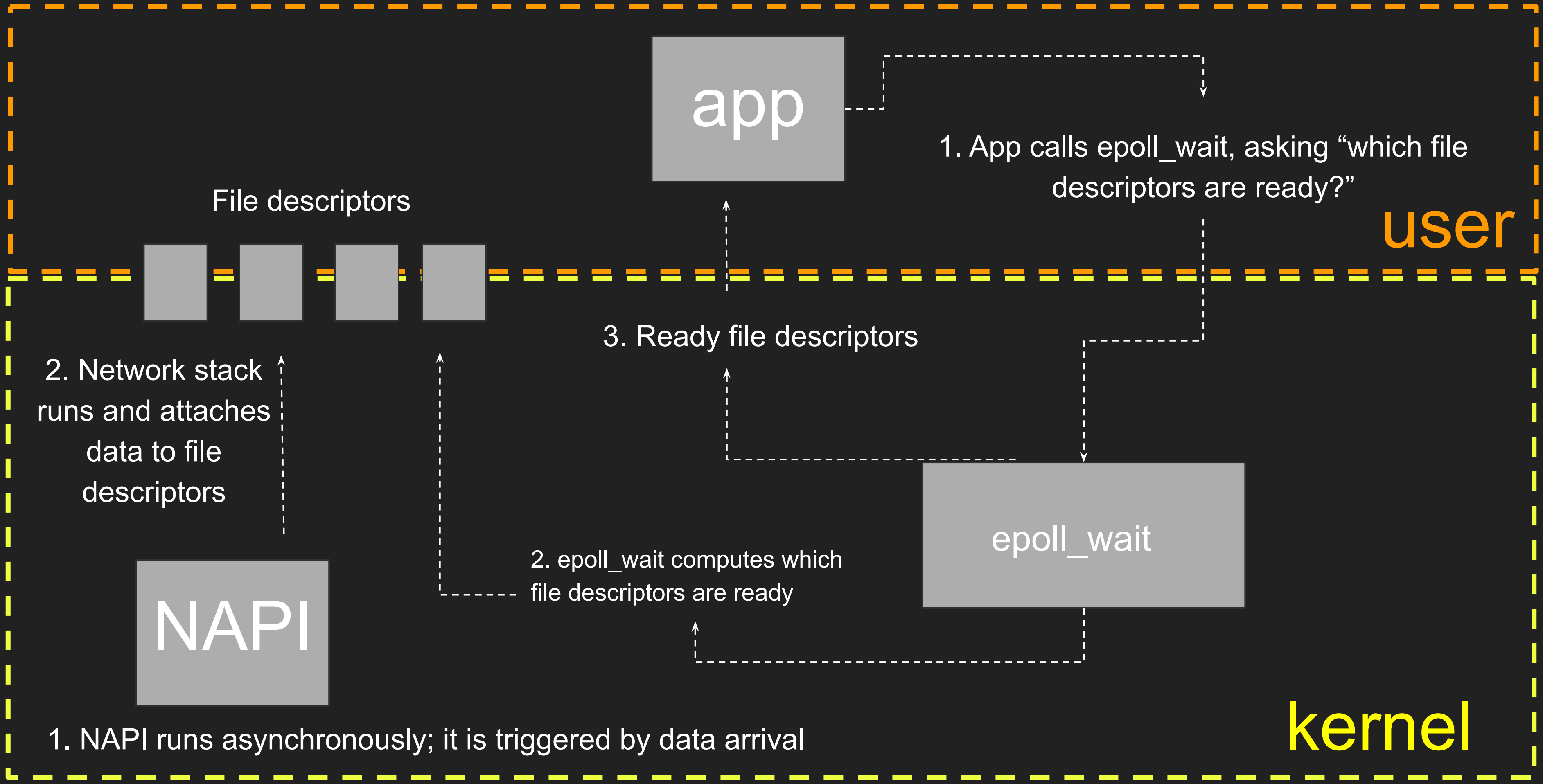
Hi, my name is Joe.

I work at Fastly.

My opinions are my own.

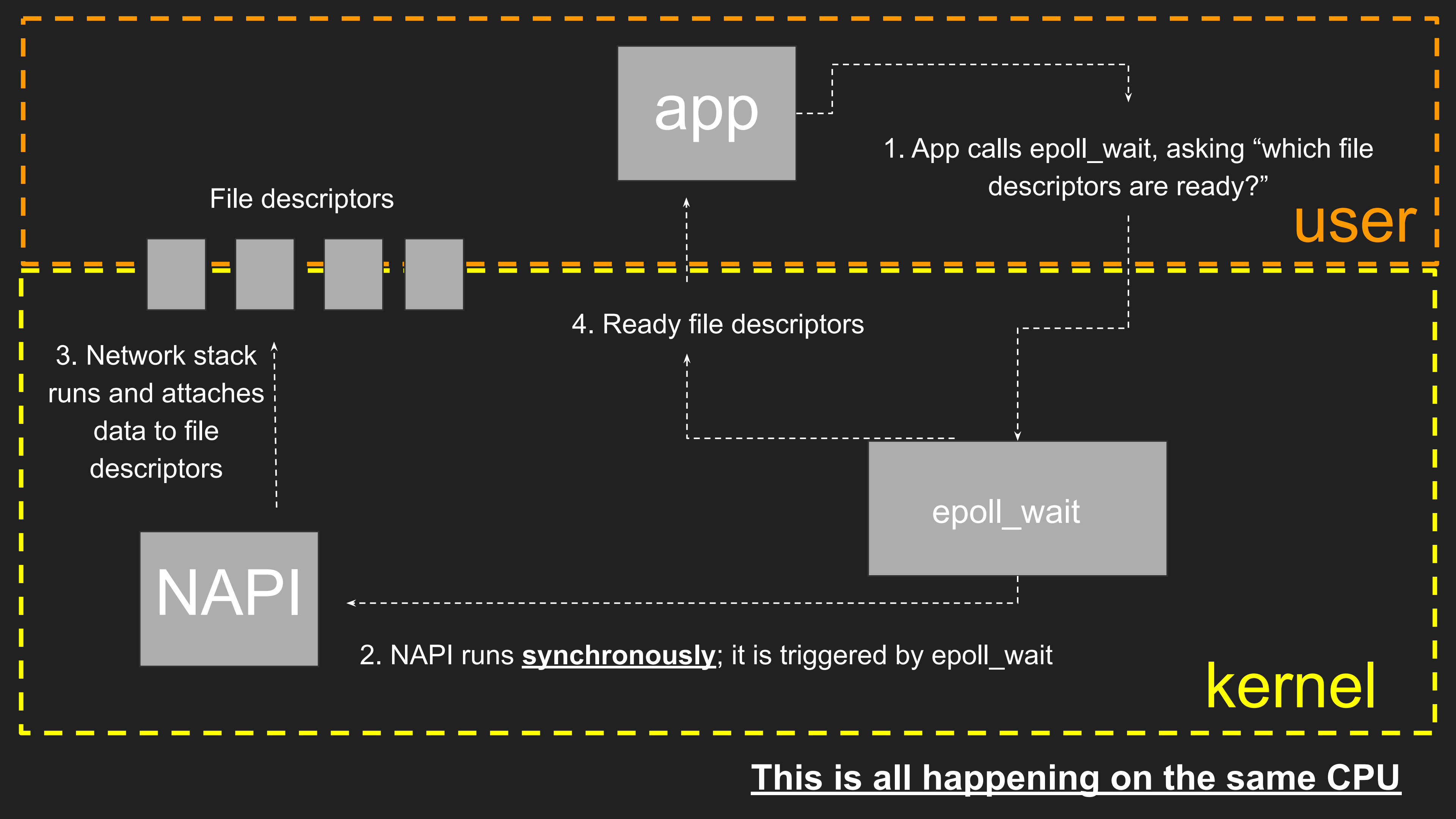
Netdev 0x18 Tutorial

Real world tips, tricks, and notes of using epoll-based busy polling to reduce latency



We’ll assume this all happens on the same CPU

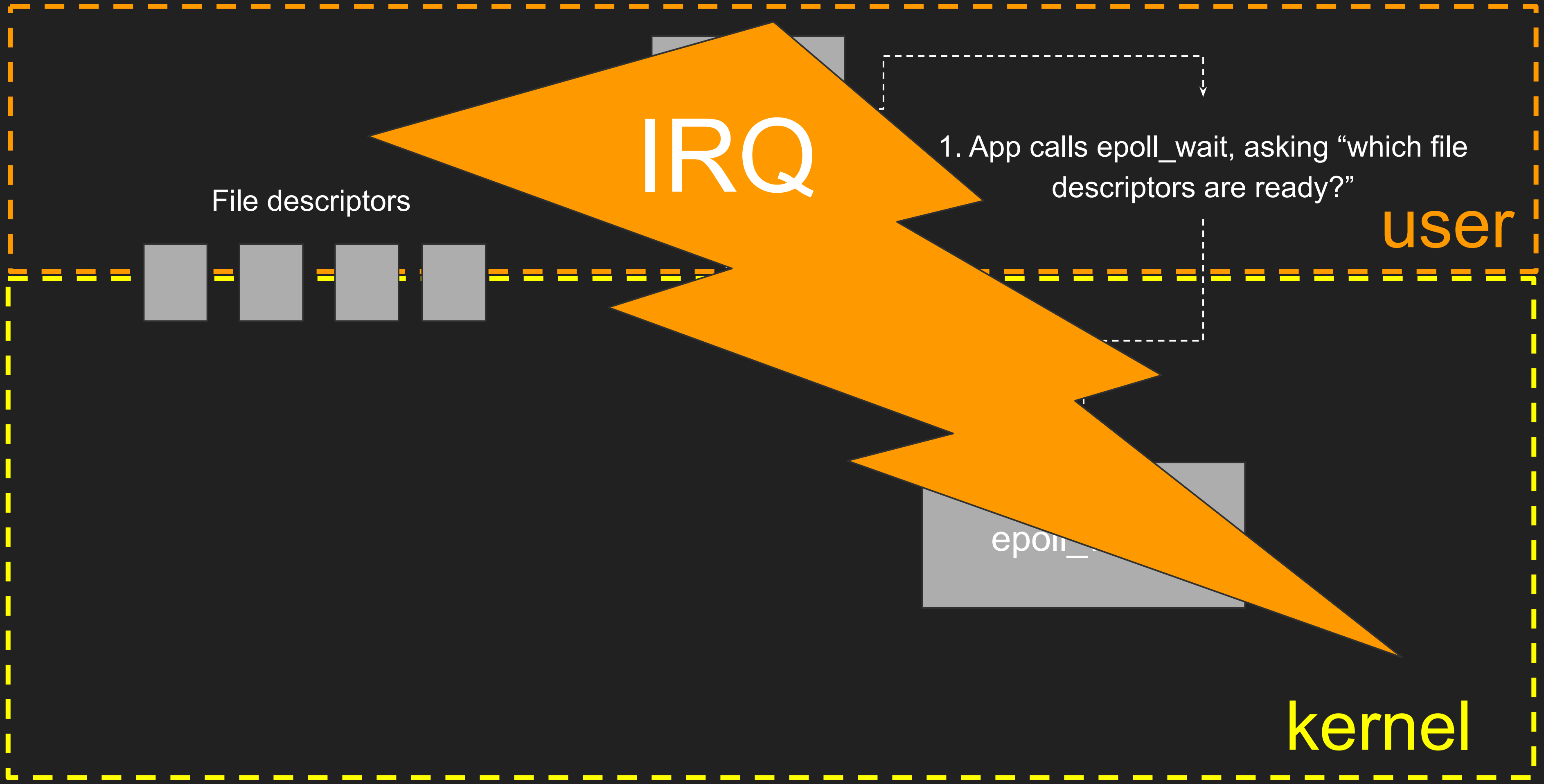
epoll + SO_INCOMING_NAPI_ID


app
1. App calls epoll_wait, asking “which file descriptors are ready?”
File descriptors
user
4. Ready file descriptors
3. Network stack runs and attaches data to file descriptors
epoll_wait
NAPI
2. NAPI runs **synchronously**; it is triggered by epoll_wait
kernel
This is all happening on the same CPU

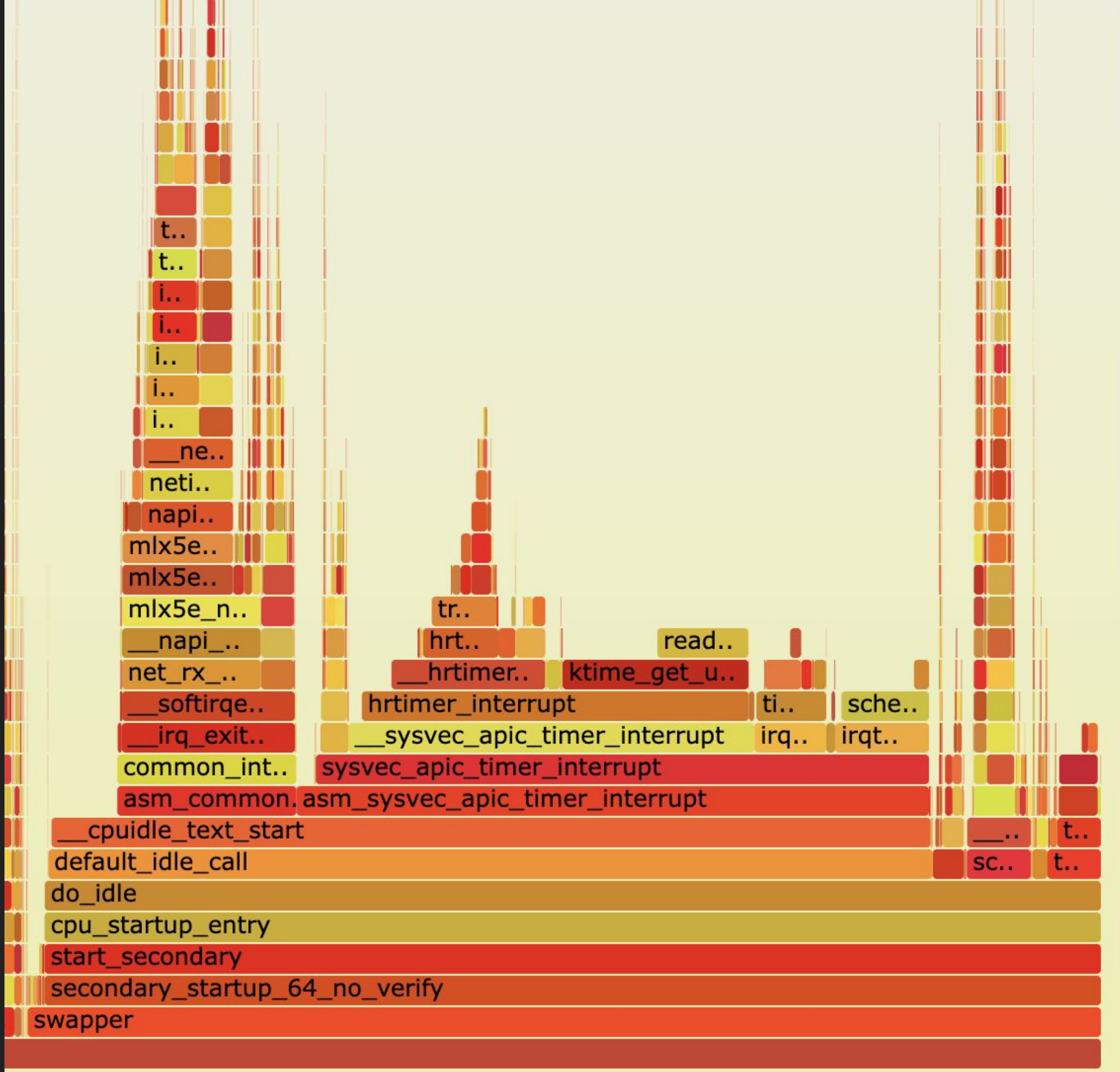

EVERYBODY HAS A PLAN .. UNTIL
THEY GET PUNCHED IN THE MOUTH!

However…

IRQs are still generated



<u>This is all happening on the same CPU</u>


t..
t..
i..
i..
i..
i..
i..
__ne..
neti..
napi..
mlx5e..
mlx5e..
mlx5e_n..
__napi_..
net_rx_..
__softirqe..
__irq_exit..
common_int..
asm_common.
tr..
hrt..
__hrtimer..
ktime_get_u..
read..
hrtimer_interrupt
ti..
sche..
__sysvec_apic_timer_interrupt
irq..
irqt..
sysvec_apic_timer_interrupt
asm_sysvec_apic_timer_interrupt
__cpuidle_text_start
__..
t..
default_idle_call
sc..
t..
do_idle
cpu_startup_entry
start_secondary
secondary_startup_64_no_verify
swapper

Worst case:

- Burning 100% CPU when there’s no work
- When there is work, IRQs interfere

IRQ suspension exists to help solve this problem.


SAY WHAT?
imgflip.com

3 part problem

1. Inherent tradeoff between CPU usage and network processing latency.

   Spend more CPU cycles to reduce latency

   Or

   Increase latency to save CPU cycles.

2. Device IRQs can interfere with network processing leading to efficiency loss due to suboptimal CPU cache usage.

3. Existing mechanisms are:
   - device specific (HW IRQ coalescing)
   - Too coarse grained (NIC wide)

And

Picking the “right” values is hard with dynamic network traffic load.

Two existing NIC wide mechanisms I stumbled upon are:

- defer_hard_irqs
- gro_flush_timeout

defer_hard_irqs

```
commit 6f8b12d661d09b488b9ac879b8eafbd2cc4a1450
Author: Eric Dumazet <edumazet@google.com>
Date:   Wed Apr 22 09:13:27 2020 -0700

    net: napi: add hard irqs deferral feature
```

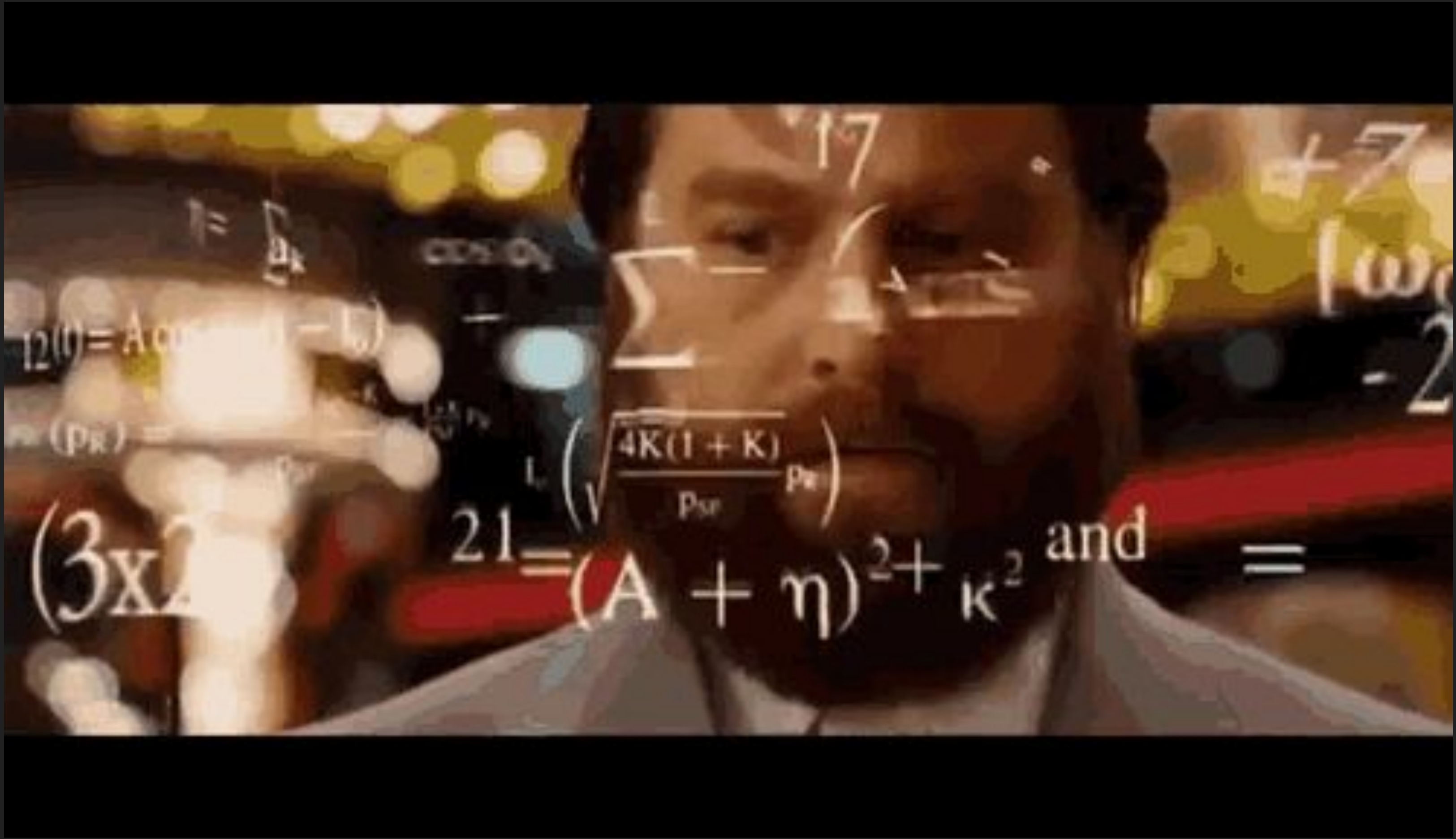
```
This feature also can be used to work around some non-optimal NIC irq
coalescing strategies.

Having the ability to insert XX usec delays between each napi->poll()
can increase cache efficiency, since we increase batch sizes.

It also keeps serving cpus not idle too long, reducing tail latencies.
```

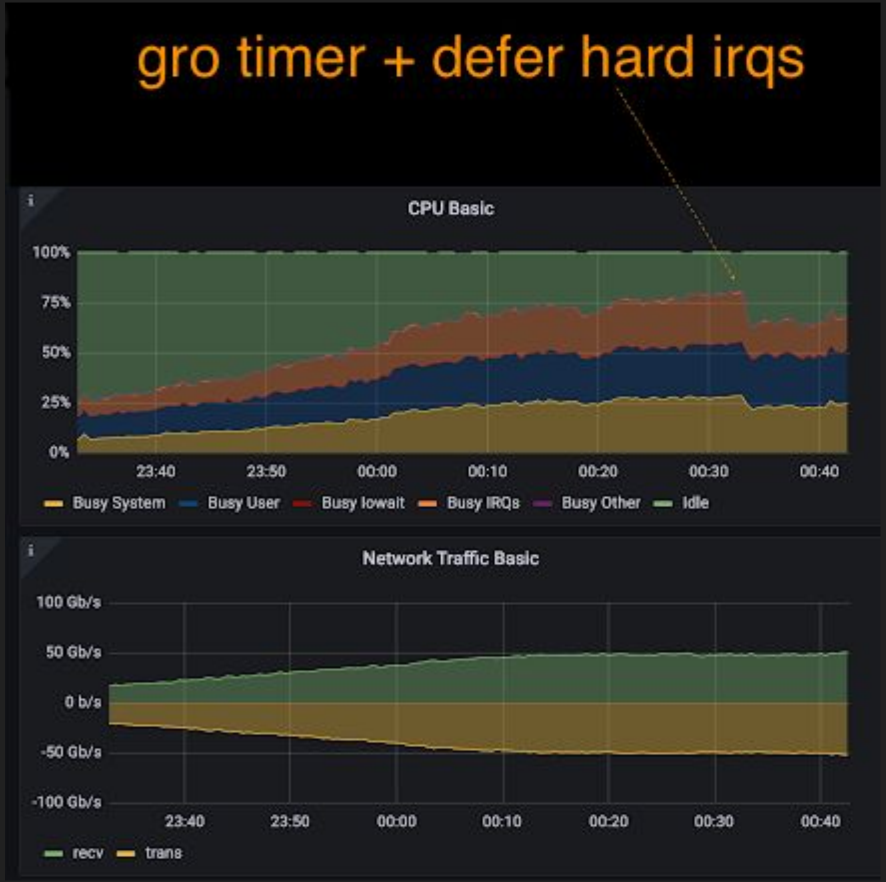
gro_flush_timeout

```
commit 3b47d30396bae4f0bd1ff0dbcd7c4f5077e7df4e
Author: Eric Dumazet <edumazet@google.com>
Date:   Thu Nov 6 21:09:44 2014 -0800

    net: gro: add a per device gro flush timer

    Tuning coalescing parameters on NIC can be really hard.
```

gro_flush_timeout

```
Setting a timer of 2000 nsec is enough to increase GRO packet sizes
and reduce number of ACK packets. (811/19.2 = 42)

Receiver performs less calls to upper stacks, less wakes up.
This also reduces cpu usage on the sender, as it receives less ACK
packets.

Note that reducing number of wakes up increases cpu efficiency, but can
decrease QPS, as applications wont have the chance to warmup cpu caches
doing a partial read of RPC requests/answers if they fit in one skb.
```


4K(1 + K)
21
(A + η)²+ κ²
and
(3x2



Changing these values on a machine shows a clear connection with CPU usage.

It is extremely difficult to choose the “right” value in environments with dynamic network traffic load.

And

NIC-wide settings affect all apps using the NIC.

A Fastly computer looks like this


DETOUR

From last year:

I am *hoping* to work on:

napi_defer_hard_irqs
gro_flush_timeout

per NAPI

(via netdev-genl hopefully?)


WELL WELL WELL
imgflip.com

Before we talk about IRQ suspension, we first need to touch on per-NAPI config settings.

per-NAPI config

- defer_hard_irq and gro_flush_timeout become configurable per NAPI (via netlink)

- Allows for other settings to be configured per NAPI (like IRQ suspension)

- Solves the interface-wide config problem

per-NAPI config

Several drivers support per-NAPI config:

- ena
- gve
- bnxt
- tg3
- tsnep
- e1000 / e1000e
- ice
- igc
- mlx4
- mlx5
- fbnic
- virtio_net
- igb (soon?)

per-NAPI config

However …

per-NAPI config

per-NAPI config settings on those devices do not *necessarily* persist between NAPI teardown and creation (for example: queue resize).

Only a subset of those drivers support persistent NAPI config.

Persistent per-NAPI config

Persistent per-NAPI config needs to be supported by the driver with a call to netif_napi_add_config. Check kernel docs.

Currently supported by:

- bnxt
- ice
- idpf
- mlx4
- mlx5
- virtio_net

Soon to be supported by:

- igb (i think)

Persistent per-NAPI config is not required for setting defer_hard_irqs or irq_suspend_timeout, but it is helpful.

I hope to add persistent per-NAPI config support to more drivers as I have time.

An example

1. Create a custom RSS context which distributes flows to queues 0-7.

2. Attach a filtering rule to steer incoming tcp4 flows on port 80 to the custom context.

3. Netlink is used to configure the NAPIs for queues 0-7 to defer hardware IRQs.

Example, steps 1 & 2:

```
#!/bin/bash

DEV=eth0

# create a custom RSS context which sends all flows to queue 0-7, this is
# context 1
sudo ethtool -X $DEV weight 1 1 1 1 1 1 1 1 context new

# add a rule to send all tcp4 flows with a dst-port of 80 to the queues in
# RSS context 1 (e.g. queues 0-7)
sudo ethtool -U $DEV flow-type tcp4 dst-port 80 context 1
```

Example, step 3:

Get the NAPI ID associated with queue 0:

```
$ ./tools/net/ynl/pyynl/cli.py \
--spec Documentation/netlink/specs/netdev.yaml \
--do queue-get \
--json='{"ifindex": 7, "id": 0, "type": "rx"}'
```

```
{'id': 0, 'ifindex': 7, 'napi-id': 8392, 'type': 'rx'}
```

Example, step 3:

Get current settings for NAPI ID 8392:

```
$ ./tools/net/ynl/pyynl/cli.py \
--spec Documentation/netlink/specs/netdev.yaml \
--do napi-get --json='{"id": 8392}'
```


```
{'defer-hard-irqs': 0,
 'gro-flush-timeout': 0,
 'id': 8392,
 'ifindex': 7,
 'irq': 327}
```

Example, step 3:

Set custom settings for NAPI ID 8392:

```
$ ./tools/net/ynl/pyynl/cli.py \
--spec Documentation/netlink/specs/netdev.yaml \
--do napi-set \
--json='{"id": 8392, "defer-hard-irqs": 10,
"gro-flush-timeout": 20000}'
```

And repeat with queues 1-7 using the python CLI as shown….

Or:

Programmatically with libynl, which now has a make target.

Worth noting:

Persistent per-NAPI configuration persists NAPI IDs, which is helpful for applications using SO_INCOMING_NAPI_ID.

So, now that we can configure things on a per-NAPI basis it is possible to add new per-NAPI features…

Like IRQ suspension.

What is IRQ suspension?

It is a mechanism which allows userland apps to drive network processing (via epoll) without interruption from device IRQs until:

1. Polling for network data on a NAPI finds no data, or

2. The irq suspension timeout is triggered

Think of it as a way to balance CPU consumption with network processing latency:

- When network traffic is high, IRQs are suspended so the userland app can run without interruption.
- When network traffic is low, IRQs are automatically re-enabled allowing userland apps to sleep, saving CPU cycles.

The cover letter of the series provides a lot of detail on the implementation and performance results of the code available in kernel 6.13+.

The paper which motivated the work was written by Peter Cai and Martin Karsten from the University of Waterloo.

The paper provides the background, measurement methodology, and comparison with other existing mechanisms.

How and when can IRQ suspension be used?

Check the kernel documentation.

Minimum requirements

1. The userland application is a network dominant application which uses epoll.
2. The NIC driver supports per-NAPI configuration configurable via netlink. Persistence is not necessary, but a nice to have.
3. Kernel 6.13+ is being used.

Optional

1. The userland application *already* uses SO_INCOMING_NAPI_ID to distribute incoming connections to worker threads.

   If not, it'll need to be modified to do so.

2. NIC hardware supports ntuple filters to steer network flows to the queues which will use IRQ suspension.

   Optional, but helpful if the system runs many different apps and only a subset of RX queues will be dedicated to the network dominant application.

Implementation

1. Optional: ntuple filters to direct flows to specific queues.

2. Userland application is modified to use:
   a. epoll_wait, noting that:
      i. max_events controls the maximum number of events userland will process per call and is closely related to the irq-suspend-timeout
      ii. A timeout of -1 can be used; if events are found they are returned to userland, otherwise the application can sleep saving CPU cycles.

   b. SO_INCOMING_NAPI_ID to distribute incoming connections to epoll loops such that each epoll loop only has incoming connections from the same NAPI ID.

Implementation

3. The epoll EPIOCSPARAMS ioctl is used for each epoll loop to set:
   a. busy_poll_usecs = 0
   b. busy_poll_budget = 64 (or less)
   c. prefer_busy_poll = true

4. Using the python CLI or libynl, for each NAPI associated with a queue where a relevant network flow will arrive set:
   a. defer-hard-irqs to a low value (e.g. 10)
   b. gro-flush-timeout to a low value (e.g. 20,000)
   c. irq-suspend-timeout to the maximum time in nanoseconds that IRQs can be suspended for – typically the maximum time the application needs to process events retrieved from epoll_wait.

Implementation Examples

Implementation example

A simple epoll_wait busy poll example which uses the EPIOCSPARAMS ioctl and libynl to set irq-suspend-timeout, see the selftest in the kernel:

https://web.git.kernel.org/pub/scm/linux/kernel/git/netdev/net-next.git/tree/tools/testing/selftests/net/busy_poller.c

Implementation example

Memcached already uses epoll_wait and supports SO_INCOMING_NAPI_ID (step 2).

Martin wrote a patch to add support for the epoll EPIOCSPARAMS ioctl (step 3), see:

https://raw.githubusercontent.com/martinkarsten/irqsuspend/main/patches/memcached.patch

Note that the irq-suspend-timeout and other parameters must be set manually using the python CLI (step 4).

Implementation example

For an example implementation that is in progress which uses libynl and supports multiple interfaces, see:

https://github.com/h2o/h2o/pull/3462

What is the performance impact?

Performance impact

Full details about test environment, test scenarios, versions of everything, patches, scripts, etc are covered in detail in the cover letter for the series.

Performance impact

But the high level summary:

- We tested memcached with mutilate
- Different configurations of defer-hard-irqs, busy polling, and “regular” NAPI processing
- Varying traffic levels

Performance impact

Low network load - full data

testcase	load	qps	avglat	95%lat	99%lat	cpu	cpq	ipq
base	200K	199946	112	239	416	26	12973	11343
defer10	200K	199971	54	124	142	29	19412	17460
defer20	200K	199986	60	130	153	26	15644	14095
defer50	200K	200025	79	144	182	23	12122	11632
defer200	200K	199999	164	254	309	19	8923	9635
fullbusy	200K	199998	46	118	133	100	43658	23133
napibusy	200K	199983	100	237	277	56	24840	24716
suspend0	200K	200020	105	249	432	30	14264	11796
suspend10	200K	199950	53	123	141	32	19518	16903
suspend20	200K	200037	58	126	151	30	16426	14736
suspend50	200K	199961	73	136	177	26	13310	12633
suspend200	200K	199998	149	251	306	21	9566	10203

Performance impact

Low network load - selected data

testcase	load	qps	avglat	95%lat	99%lat	cpu	cpq	ipq
base	200K	199946	112	239	416	26	12973	11343
fullbusy	200K	199998	46	118	133	100	43658	23133
suspend10	200K	199950	53	123	141	32	19518	16903

- suspend10 uses much less CPU than full busy polling for comparable latency

- suspend10 uses slightly more CPU than regular NAPI processing, but at ~half the latency

Performance impact

Max network load - full data

testcase	load	qps	avglat	95%lat	99%lat	cpu	cpq	ipq
base	MAX	1037654	4184	5453	5810	100	8411	7938
defer10	MAX	905607	4840	6151	6380	100	9639	8431
defer20	MAX	986463	4455	5594	5796	100	8848	8110
defer50	MAX	1077030	4000	5073	5299	100	8104	7920
defer200	MAX	1040728	4152	5385	5765	100	8379	7849
fullbusy	MAX	1247536	3518	3935	3984	100	6998	7930
napibusy	MAX	1136310	3799	7756	9964	100	7670	7877
suspend0	MAX	1057509	4132	5724	6185	100	8253	7918
suspend10	MAX	1215147	3580	3957	4041	100	7185	7944
suspend20	MAX	1216469	3576	3953	3988	100	7175	7950
suspend50	MAX	1215871	3577	3961	4075	100	7181	7949
suspend200	MAX	1216882	3556	3951	3988	100	7175	7955

Performance impact

Max network load - selected data

testcase	load	qps	avglat	95%lat	99%lat	cpu	cpq	ipq
base	MAX	1037654	4184	5453	5810	100	8411	7938
fullbusy	MAX	1247536	3518	3935	3984	100	6998	7930
suspend10	MAX	1215147	3580	3957	4041	100	7185	7944

- suspend10 has ~14% more application queries per second than regular NAPI processing *and* better latency.

- suspend10 uses the same CPU as full busy polling with comparable latency.

Performance impact

Summary

- At low load IRQ suspension has:
    - comparable latency to full busy polling, but with *much less* CPU usage.
    - Slightly higher CPU than regular NAPI processing, but half the latency.

- At maximum load IRQ suspension has:
    - better processing efficiency than regular NAPI processing (higher application QPS).
    - Comparable latency to full busy polling.

In conclusion:

IRQ suspension provides a mechanism for reducing CPU usage at low network load while providing low latency at maximum network load, automatically.

?