

Advanced Network Programming (ANP)

XB_0048

Linux Networking

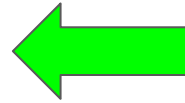
Animesh Trivedi
Autumn 2020, Period 1

Layout of upcoming lectures - Part 1

Sep 1st, 2020 (today): ~~Introduction and networking concepts~~

Sep 3rd, 2020 (this Tuesday): ~~Networking concepts (continued)~~

Sep 8th, 2020 : *Linux networking internals*



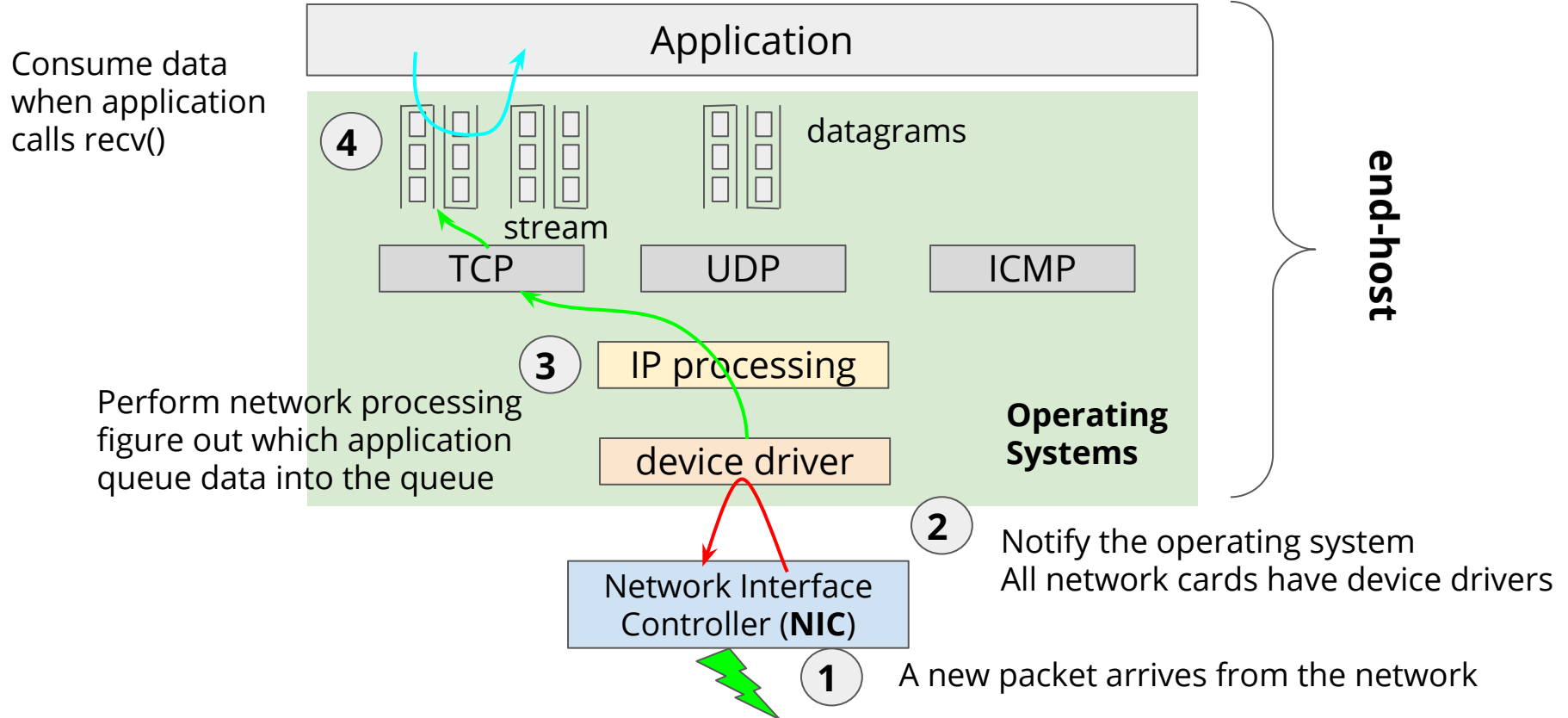
- MTU and link efficiency
- TSO, LRO, GRO
- TCP offload engine
- Stateless and stateful offloads

Sep 10th 2020: *Multicore scalability*

Sep 15th 2020: *Userspace networking stacks*

Sep 17th 2020: *Introduction to RDMA networking*

A packet's journey - (simplified) Receiving path



Linux Code Layout

```
arch      cmake-build-debug  crypto      firmware   ipc         lib          Makefile    Module.symvers  scripts  tags  vmlinux
block     CMakeLists.txt          debian      fs          Kbuild      LICENSES     mm          net             security  tools  vmlinux-gdb.py
built-in.o COPYING          Documentation include  Kconfig     linux-4.15.0.tar.gz modules.builtin  README   sound  usr    vmlinux.o
certs     CREDITS                  drivers     init        kernel      MAINTAINERS  modules.order  samples  System.map  virt   I
```

```
linux/drivers/net$
```

All networking device drivers are here

```
linux/drivers/net/ethernet$
```

Ethernet drivers here

```
linux/net/ipv4$
```

IPv4, TCP, UDP implementation here

```
linux/net/core$
```

Many common helper routines here (buffers, sockets)

What NIC / driver do you use?

```
atr@atr-XPS-13:~$ ifconfig
enx9cebe8cd8f11: flags=4099<UP,BROADCAST,MULTICAST> mtu 1500
    ether 9c:eb:e8:cd:8f:11 txqueuelen 1000 (Ethernet)
    RX packets 0 bytes 0 (0.0 B)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 0 bytes 0 (0.0 B)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING> mtu 8192
    inet 127.0.0.1 netmask 255.0.0.0
    inet6 ::1 prefixlen 128 scopeid 0x10<host>
    loop txqueuelen 1000 (Local Loopback)
    RX packets 4268 bytes 617794 (617.7 KB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 4268 bytes 617794 (617.7 KB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

virbr0: flags=4099<UP,BROADCAST,MULTICAST> mtu 1500
    inet 192.168.122.1 netmask 255.255.255.0 broadcast 192.168.122.255
    ether 52:54:00:6f:f9:16 txqueuelen 1000 (Ethernet)
    RX packets 0 bytes 0 (0.0 B)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 0 bytes 0 (0.0 B)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

wlp2s0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 192.168.1.81 netmask 255.255.255.0 broadcast 192.168.1.255
    inet6 fe80::c21:691e:bcec:c511 prefixlen 64 scopeid 0x20<link>
    ether 9c:b6:d0:97:92:47 txqueuelen 1000 (Ethernet)
    RX packets 18033574 bytes 11070371827 (11.0 GB)
    RX errors 0 dropped 8 overruns 0 frame 0
    TX packets 7997937 bytes 5210819036 (5.2 GB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
```

Active NIC interfaces

ethtool -i wlp2s0 (info)

```
atr@atr-XPS-13:~$ ethtool -i wlp2s0
```

```
driver: ath10k_pci
version: 4.15.0-1081-oem
firmware-version: WLAN.RM.4.4.1-00079-QCARMSWPZ-1
expansion-rom-version:
bus-info: 0000:02:00.0
supports-statistics: yes
supports-test: no
supports-eeprom-access: no
supports-io-ports: no
supports-pci: no
```

driver in use for this device

```
atr@atr-XPS-13:~$ lspci -v | grep "02:00.0"
```

```
02:00.0 Network controller: Qualcomm Atheros QCA6174 802.11ac Wireless Network Adapter (rev 32)
```

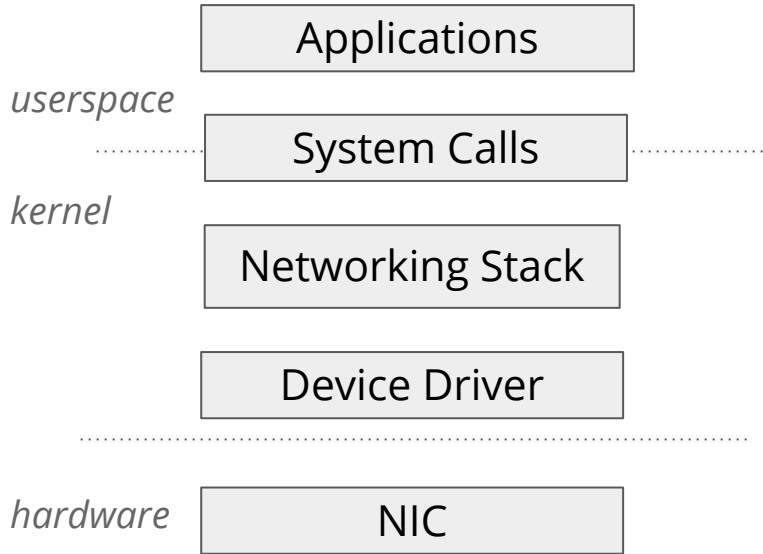
Too much information?



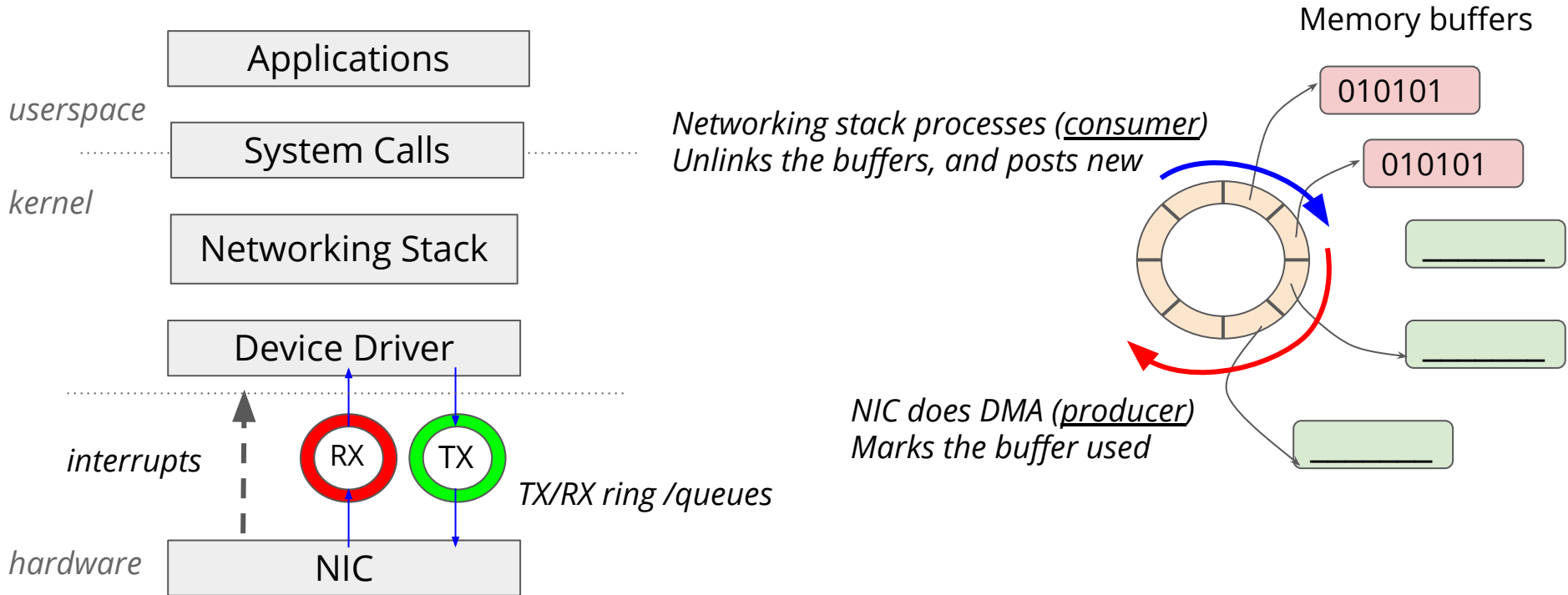
- Linux kernel source code, function or structure names, command names or their parameter are **not a part of your exam**
- Information on various external links is **not a part of your exam**
 - they are put here for your reading, understand, and reference, if you are interested

However, various high level ideas and concepts in networking (which are on these slides) are **part of your exam**. *For example, what is NAPI? Why it was designed?*

The device interface



The device interface: Rings



This **producer-consumer ring pattern** is a very common data structure (and pattern) in many areas in OS, virtualization, storage and networking designs

Linux Tool: ethtool -g

```
ETHTOOL(8)                System Manager's Manual                ETHTOOL(8)

NAME
    ethtool - query or control network driver and hardware settings

SYNOPSIS
    ethtool devname

    ethtool -h|--help

    ethtool --version

    ethtool -a|--show-pause devname

    ethtool -A|--pause devname [autoneg on|off] [rx on|off] [tx on|off]

    ethtool -c|--show-coalesce devname

    ethtool -C|--coalesce devname [adaptive-rx on|off] [adaptive-tx on|off]
    [rx-usecs N] [rx-frames N] [rx-usecs-irq N] [rx-frames-irq N]
    [tx-usecs N] [tx-frames N] [tx-usecs-irq N] [tx-frames-irq N]
    [stats-block-usecs N] [pkt-rate-low N] [rx-usecs-low N]
    [rx-frames-low N] [tx-usecs-low N] [tx-frames-low N]
    [pkt-rate-high N] [rx-usecs-high N] [rx-frames-high N]
    [tx-usecs-high N] [tx-frames-high N] [sample-interval N]

    ethtool -g|--show-ring devname

    ethtool -G|--set-ring devname [rx N] [rx-mini N] [rx-jumbo N] [tx N]
```

```
-g --show-ring
    Queries the specified network device for rx/tx ring parameter
    information.

-G --set-ring
    Changes the rx/tx ring parameters of the specified network de-
    vice.

    rx N    Changes the number of ring entries for the Rx ring.

    rx-mini N
    Changes the number of ring entries for the Rx Mini ring.

    rx-jumbo N
    Changes the number of ring entries for the Rx Jumbo ring.

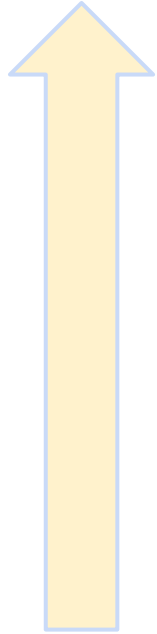
    tx N    Changes the number of ring entries for the Tx ring.
```

```
atr@evelyn:~$ ethtool -g enp0s25
```

```
Ring parameters for enp0s25:
Pre-set maximums:
RX:                               4096
RX Mini:                           0
RX Jumbo:                           0
TX:                               4096
Current hardware settings:
RX:                               256
RX Mini:                           0
RX Jumbo:                           0
TX:                               256
```

```
atr@evelyn:~$
```

Linux Packet Receive Path



Copy and consumption by an application

Queueing up at a [socket receive queue](#)

Local TCP/UDP processing

Local IP processing

[Socket Kernel Buffer \(SKBs\)](#)

Networking stack: ["bottom-half"](#) (softirqs)

Networking stack: ["top-half"](#)

Hardware interrupt

Does DMA to the next free ring buffer address

NIC receives a packet

The Source Code is all we have ;)

Plenty of comments !

Clip slide

```
* struct net_device - The DEVICE structure.
*
* Actually, this whole structure is a big mistake. It mixes I/O
* data with strictly "high-level" data, and it has to know about
* almost every data structure used in the INET module.

/* Accept zero addresses only to limited broadcast;
* I even do not know to fix it or not. Waiting for complains :-)

/* An explanation is required here, I think.
* Packet length and doff are validated by header prediction,

* Really tricky (and requiring careful tuning) part of algorithm
* is hidden in functions tcp_time_to_recover() and
tcp_xmit_retransmit_queue().

/* skb reference here is a bit tricky to get right, since
* shifting can eat and free both this skb and the next,
* so not even _safe variant of the loop is enough.

/* Here begins the tricky part :
* We are called from release_sock() with :

/* This is TIME_WAIT assassination, in two flavors.
* Oh well... nobody has a sufficient solution to this
* protocol bug yet.

BUG(); /* "Please do not press this button again." */

/* The socket is already corked while preparing it. */
/* ... which is an evident application bug. --ANK */

/* Ugly, but we have no choice with this interface.
* Duplicate old header, fix ihl, length etc.

* Parse and mangle SNMP message according to mapping.
* (And this is the fucking 'basic' method).

/* 2. Fixups made earlier cannot be right.
*      If we do not estimate RTO correctly without them,
*      all the algo is pure shit and should be replaced
*      with correct one. It is exactly, which we pretend to do.

/* OK, ACK is valid, create big socket and
* feed this segment to it. It will repeat all
* the tests. THIS SEGMENT MUST MOVE SOCKET TO
* ESTABLISHED STATE. If it will be dropped after
* socket is created, wait for troubles.

* packets force peer to delay ACKs and calculation is correct too.
* The algorithm is adaptive and, provided we follow specs, it
* NEVER underestimate RTT. BUT! If peer tries to make some clever
* tricks sort of "quick acks" for time long enough to decrease RTT
* to low value, and then abruptly stops to do it and starts to delay
* ACKs, wait for troubles.
```

Linux Packet Receive Path

A networking device in Linux is represented by `struct net_device`

- Contains functions for all device I/O activities, management, bringing up and down the device
`struct net_device_ops`
- Very very large structure
- Makes the contract between a device driver and the rest of the networking stack

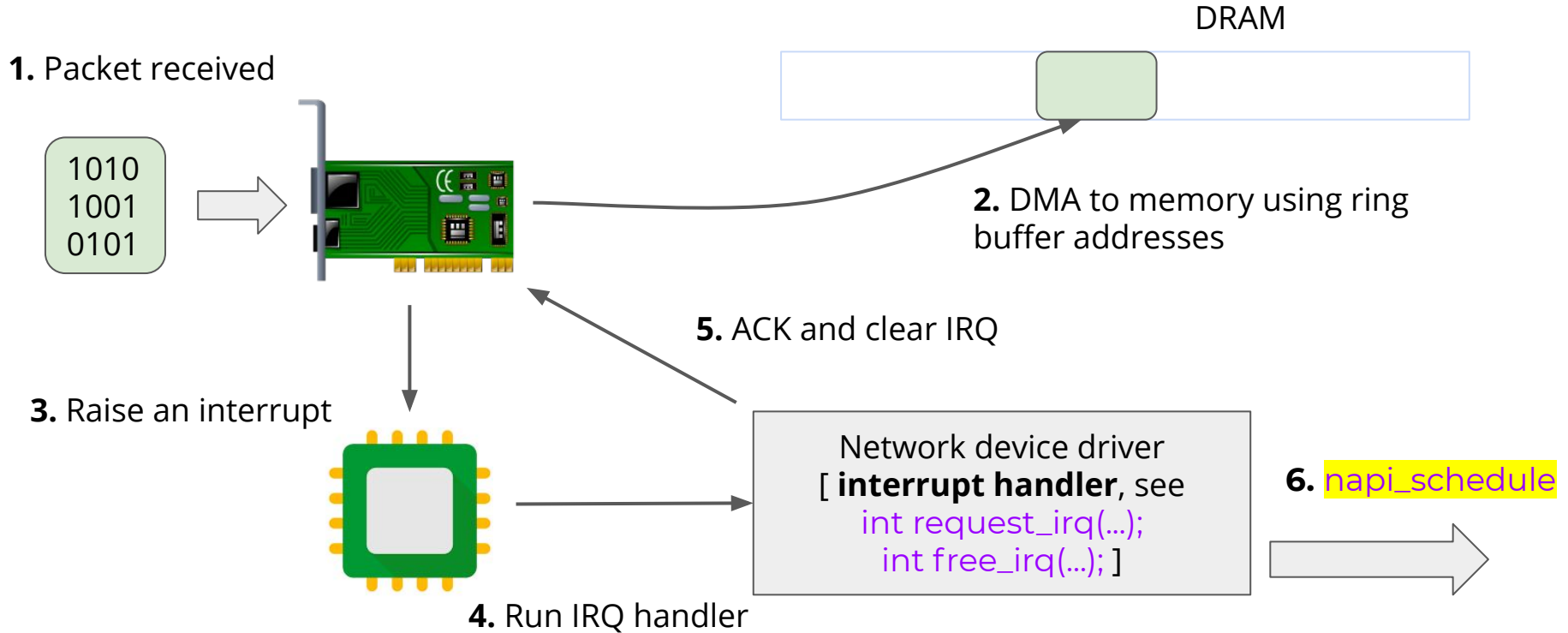
<https://elixir.bootlin.com/linux/latest/source/include/linux/netdevice.h>

```
/**
 * struct net_device - The DEVICE structure.
 *
 * Actually, this whole structure is a big mistake. It mixes I/O
 * data with strictly "high-level" data, and it has to know about
 * almost every data structure used in the INET module.
 *
 * @name: This is the first field of the "visible" part of this structure
 * (i.e. as seen by users in the "Space.c" file). It is the name
 * of the interface.
 *
 * @name_node: Name hashlist node
```

```
struct net_device {
    char                name[IFNAMSIZ];
    struct netdev_name_node *name_node;
    struct dev_ifalias  __rcu *ifalias;
    /*
     * I/O specific fields
     * FIXME: Merge these and struct ifmap into one
     */
    unsigned long        mem_end;
    unsigned long        mem_start;
    unsigned long        base_addr;
    int                  irq;

    /*
     * Some hardware also needs these fields (state, dev_list,
     * napi_list, unreg_list, close_list) but they are not
     * part of the usual set specified in Space.c.
     */
}
```

Linux Packet Receive Path



What is New API (NAPI)? (so much innovation in naming)

There are only two hard things in Computer Science: cache invalidation and naming things.

-- Phil Karlton

New API or NAPI is the implementation of interrupt mitigation technique that we looked earlier (all info in: `struct napi_struct`)

- By default interrupt: check if NAPI is already requested to be scheduled
- In the NAPI processing:
 - Interrupts on the NIC are disabled
 - Process certain number of “weight” packets in one go
 - Poll the device driver (by calling a driver provided function) to check if there are more packets ready for processing
 - `void netif_napi_add(struct net_device *dev, struct napi_struct *napi, int (*poll)(struct napi_struct *, int), int weight);`
 - If not enough packets available, then yield
 - The driver will enable the interrupt-driven notification again on the NIC

What does `napi_schedule` do?

```
/**
 * napi_schedule - schedule NAPI poll
 * @n: napi context
 *
 * Schedule NAPI poll routine to be called if it is not already
 * running.
 */
static inline void napi_schedule(struct napi_struct *n)
{
    if (napi_schedule_prep(n))
        napi_schedule(n);
}
```

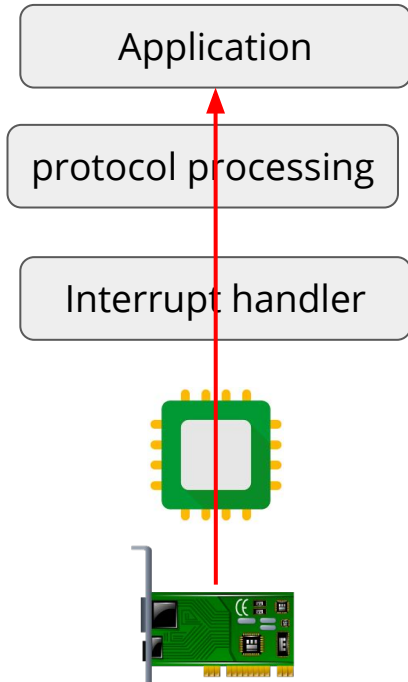
```
/**
 * __napi_schedule - schedule for receive
 * @n: entry to schedule
 *
 * The entry's receive function will be scheduled to run.
 * Consider using __napi_schedule_irqoff() if hard irq's are masked.
 */
void __napi_schedule(struct napi_struct *n)
{
    unsigned long flags;

    local_irq_save(flags);
    napi_schedule(this_cpu_ptr(&softnet_data), n);
    local_irq_restore(flags);
}
EXPORT_SYMBOL(__napi_schedule);
```

```
/* Called with irq disabled */
static inline void ____napi_schedule(struct softnet_data *sd,
                                     struct napi_struct *napi)
{
    list_add_tail(&napi->poll_list, &sd->poll_list);
    __raise_softirq_irqoff(NET_RX_SOFTIRQ);
}
```

What is a softirq?

Who is processing NAPI?

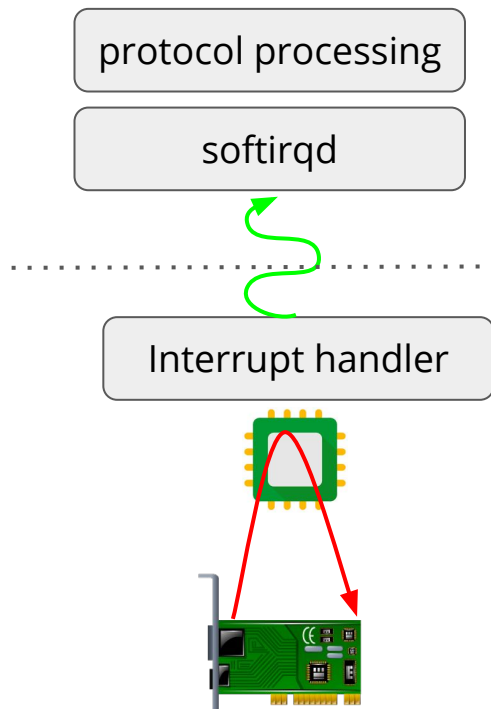


When processing an interrupt ...

- Interrupt handling is a **high priority work**
- All further interrupts are disabled (on the `_CPU_`, not just the specific device which generated the interrupt)
 - If there are more packets coming in we will miss out on them
 - If ring buffer overflow - then packets are dropped
- Many long events in the incoming packet processing - memory allocation, scheduling of application process to consume incoming data, TCP ACK generation, TCP transmission processing ... **a LOT of work**

We can not disable interrupt for this long to do all this work in the interrupt handler

Linux: Top-Half and Bottom-Half Processing



Bottom-half processing (also known as SoftIRQs)

- High priority
- Most of the protocol processing
- Have interrupts enabled (can be preempted)
- Can preempt other kernel threads, user processing

Top-Half Interrupt Processing

- Very high priority, very small
- Mostly cleaning hardware registers
- Copying out pointers
- Scheduling the bottom half, if not already there

Linux Kernel: Deferred Work Processing Framework

Linux Kernel has multiple mechanisms for deferred work processing

- **SoftIRQs:** *A specific number of predefined SoftIRQ / core for specific tasks (e.g., RX, TX, timer). **No blocking calls**. Concurrent execution of same type of softirq on different CPUs.*
- **Tasklets:** *A more flexible SoftIRQs that can be allocated and used by different kernel subsystems. No blocking calls. No concurrent execution of the same tasklet on different CPUs, serialized.*
- **WorkQueues:** *A kernel thread pool with an immediate or delayed work execution in a process context. Can have blocking calls. Managed by the kernel.*
- **Kernel Threads:** *A generic kernel thread, to be used for any purpose by the caller.*

It is very very important to always know in what context the code is executing, determines what action you can or cannot do, and how you should be locking and synchronize with other execution context inside a kernel

Preemption order and locking

| . | IRQ Handler A | IRQ Handler B | Softirq A | Softirq B | Tasklet A | Tasklet B | Timer A | Timer B | User Context A | User Context B |
|----------------|---------------|---------------|-----------|-----------|-----------|-----------|---------|---------|----------------|----------------|
| IRQ Handler A | None | | | | | | | | | |
| IRQ Handler B | SLIS | None | | | | | | | | |
| Softirq A | SLI | SLI | SL | | | | | | | |
| Softirq B | SLI | SLI | SL | SL | | | | | | |
| Tasklet A | SLI | SLI | SL | SL | None | | | | | |
| Tasklet B | SLI | SLI | SL | SL | SL | None | | | | |
| Timer A | SLI | SLI | SL | SL | SL | SL | None | | | |
| Timer B | SLI | SLI | SL | SL | SL | SL | SL | None | | |
| User Context A | SLI | SLI | SLBH | SLBH | SLBH | SLBH | SLBH | SLBH | None | |
| User Context B | SLI | SLI | SLBH | SLBH | SLBH | SLBH | SLBH | SLBH | MLI | None |

| | |
|------|--------------------------|
| SLIS | spin_lock_irqsave |
| SLI | spin_lock_irq |
| SL | spin_lock |
| SLBH | spin_lock_bh |
| MLI | mutex_lock_interruptible |

<https://www.kernel.org/doc/html/latest/kernel-hacking/locking.html>

Linux Commands

```
atr@atr-XPS-13:~$ cat /proc/softirqs
```

| | CPU0 | CPU1 | CPU2 | CPU3 | CPU4 | CPU5 | CPU6 | CPU7 |
|-----------|----------|----------|----------|----------|----------|----------|----------|----------|
| HI: | 2197884 | 1974204 | 1779038 | 2371321 | 1625381 | 35041084 | 10544874 | 2591798 |
| TIMER: | 16792747 | 18326094 | 17362586 | 17424605 | 16996167 | 23815599 | 19733631 | 20141941 |
| NET_TX: | 77 | 107 | 85 | 95 | 92 | 63 | 100 | 163 |
| NET_RX: | 6750 | 6016336 | 912360 | 446993 | 927722 | 12708 | 3165183 | 8006769 |
| BLOCK: | 50 | 58 | 124 | 836 | 116 | 60 | 63 | 71 |
| IRQ_POLL: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| TASKLET: | 1127 | 31878 | 147090 | 222415 | 27305 | 26101 | 59450 | 65613 |
| SCHED: | 25079491 | 21381249 | 18120473 | 17064896 | 16880949 | 23507174 | 18992029 | 19547016 |
| HRTIMER: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| RCU: | 12574327 | 13177430 | 12891793 | 12882834 | 12570824 | 16264599 | 13996374 | 14314682 |

```
atr@atr-XPS-13:~$
```

```
atr@atr-XPS-13:~$ ps -e | grep softirq
```

| | | |
|------|----------|-------------|
| 7 ? | 00:00:05 | ksoftirqd/0 |
| 16 ? | 00:00:12 | ksoftirqd/1 |
| 22 ? | 00:00:04 | ksoftirqd/2 |
| 28 ? | 00:00:03 | ksoftirqd/3 |
| 34 ? | 00:00:02 | ksoftirqd/4 |
| 40 ? | 00:00:05 | ksoftirqd/5 |
| 46 ? | 00:00:06 | ksoftirqd/6 |
| 52 ? | 00:00:13 | ksoftirqd/7 |

```
atr@atr-XPS-13:~$ ps -e | grep kworker
```

| | | |
|------|----------|--------------|
| 4 ? | 00:00:00 | kworker/0:0H |
| 18 ? | 00:00:00 | kworker/1:0H |
| 24 ? | 00:00:00 | kworker/2:0H |
| 30 ? | 00:00:00 | kworker/3:0H |
| 36 ? | 00:00:00 | kworker/4:0H |
| 42 ? | 00:00:00 | kworker/5:0H |
| 48 ? | 00:00:00 | kworker/6:0H |
| 54 ? | 00:00:00 | kworker/7:0H |

There is a priority order with the SoftIRQs defined, as you see in the print order: high priority, timer, network, block, polling, tasklets, scheduler, hrtimer, and RCU locks.

As usual there is a lot of information present in the /proc interface that you can explore about these execution threads.

Understand: preemption vs. priority

For more reading

I'll Do It Later: Softirqs, Tasklets, Bottom Halves, Task Queues, Work Queues and Timers

Matthew Wilcox
Hewlett-Packard Company
matthew.wilcox@hp.com

Abstract

An interrupt is a signal to a device driver that there is work to be done. However, if the driver does too much work in the interrupt handler, system responsiveness will be degraded. The standard way to avoid this problem (until Linux 2.3.42) was to use a bottom half or a task queue to schedule some work to do later. These handlers are run with interrupts enabled and lengthy processing has less impact on system response.

The work done for softnet introduced two new facilities for deferring work until later: softirqs and tasklets. They were introduced in order to achieve better SMP scalability. The existing bottom halves were reimplemented as a special form of tasklet which

1 Introduction

When writing kernel code, it is common to wish to defer work until later. There are many reasons for this. One is that it is inappropriate to do too much work with a lock held. Another may be to batch work to amortise the cost. A third may be to call a sleeping function, when scheduling at that point is not allowed.

The Linux kernel offers many different facilities for postponing work until later. Bottom Halves are for deferring work from interrupt context. Timers allow work to be deferred for at least a certain length of time. Work Queues allow work to be deferred to process context.

[Docs](#) » [Kernel Hacking Guides](#) » [Unreliable Guide To Locking](#)

Unreliable Guide To Locking

Author: Rusty Russell

Introduction

Welcome, to Rusty's Remarkably Unreliable Guide to Kernel Locking issues. This document describes the

With the wide availability of HyperThreading, and preemption in the Linux Kernel, everyone hacking on the

The Problem With Concurrency

(Skip this if you know what a Race Condition is).

In a normal program, you can increment a counter like so:

<http://www.cs.columbia.edu/~nahum/w6998/papers/2003-wilcox-softirq.pdf>
<https://www.kernel.org/doc/html/latest/kernel-hacking/locking.html>

Coming Back to NAPI Packet Processing


The main entry point for NET_RX_SOFTIRQ processing is :

`static void net_rx_action(struct softirq_action *h)`

Polls the network device, where

- Packets are build
- Pushed in the netstack for processing
- Return the “budget” consumed
- If budget consumed, or time to reschedule : break

LRO/GRO merging can happen in the driver



```
for (;;) {
    struct napi_struct *n;

    if (list_empty(&list)) {
        if (!sd_has_rps_ipi_waiting(sd) && list_empty(&repoll))
            goto out;
        break;
    }

    n = list_first_entry(&list, struct napi_struct, poll_list);
    budget -= napi_poll(n, &repoll);

    /* If softirq window is exhausted then punt.
     * Allow this to run for 2 jiffies since which will allow
     * an average latency of 1.5/HZ.
     */
    if (unlikely(budget <= 0 ||
                 time_after_eq(jiffies, time_limit))) {
        sd->time_squeeze++;
        break;
    }
}
```

In the Driver Polling Function

Data packets are build and pushed into `netif_receive_skb()`;

This does not do much, a bit of accounting, backlog processing, various tracking hooks before being delivered to the networking layer at `ip_rcv()`

But hold on, what is this `sk_buff*`?

```
/**
 * netif_receive_skb - process receive buffer from network
 * @skb: buffer to process
 *
 * netif_receive_skb() is the main receive data processing function.
 * It always succeeds. The buffer may be dropped during processing
 * for congestion control or by the protocol layers.
 *
 * This function may only be called from softirq context and interrupts
 * should be enabled.
 *
 * Return values (usually ignored):
 * NET_RX_SUCCESS: no congestion
 * NET_RX_DROP: packet was dropped
 */
int netif_receive_skb(struct sk_buff *skb)
{
    int ret;

    trace_netif_receive_skb_entry(skb);

    ret = netif_receive_skb_internal(skb);
    trace_netif_receive_skb_exit(ret);

    return ret;
}
EXPORT_SYMBOL(netif_receive_skb);
```

Socket Kernel Buffer or SKB

One of the most important data types in the Linux kernel

Represent a data packet in processing

Has headers, trailer, data, metadata, Linux specific details -- all in a single structure

Can be extremely complex
(a simplified version of this is Socket User Buffer (struct su_buff) is provided in the ANP netstack)

```
struct sk_buff {
    union {
        struct {
            /* These two members must be first. */
            struct sk_buff *next;
            struct sk_buff *prev;

            union {
                struct net_device *dev;
                /* Some protocols might use this space to store information,
                 * while device pointer would be NULL.
                 * UDP receive path is one user.
                 */
                unsigned long dev_scratch;
            };
        };
        struct rb_node rbnode; /* used in netem, ip4 defrag, and tcp stack */
        struct list_head list;
    };

    union {
        struct sock *sk;
        int ip_defrag_offset;
    };

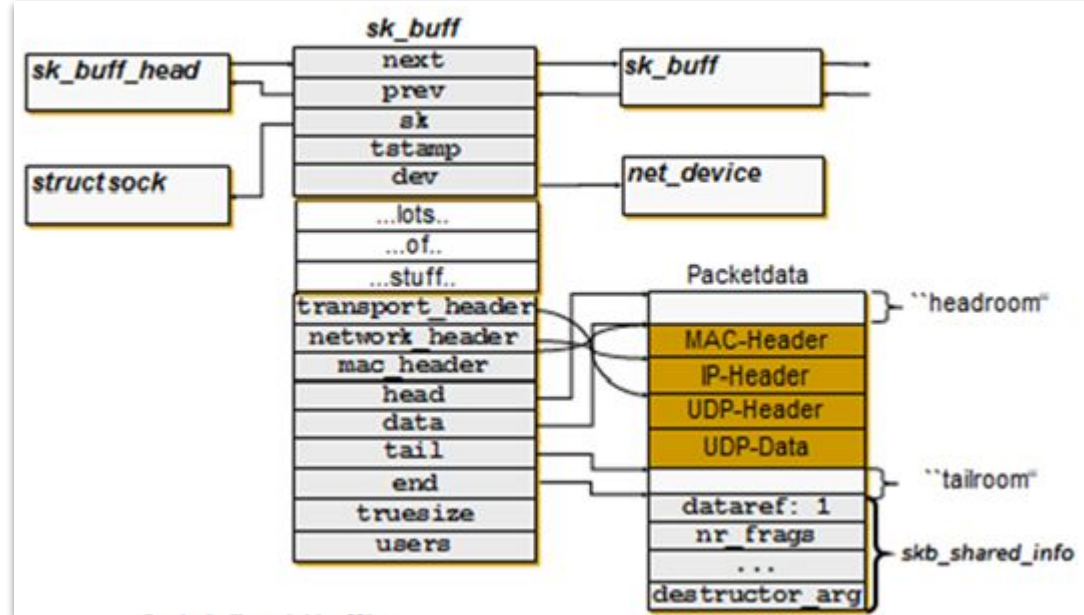
    union {
        ktime_t tstamp;
        u64 skb_mstamp_ns; /* earliest departure time */
    };
    /*
     * This is the control buffer. It is free to use for every
     * layer. Please put your private variables there. If you
     * want to keep them across layers you have to do a skb_clone()
     * first. This is owned by whoever has the skb queued ATM.
     */
    char cb[48] __aligned(8);

    union {
        struct {
```

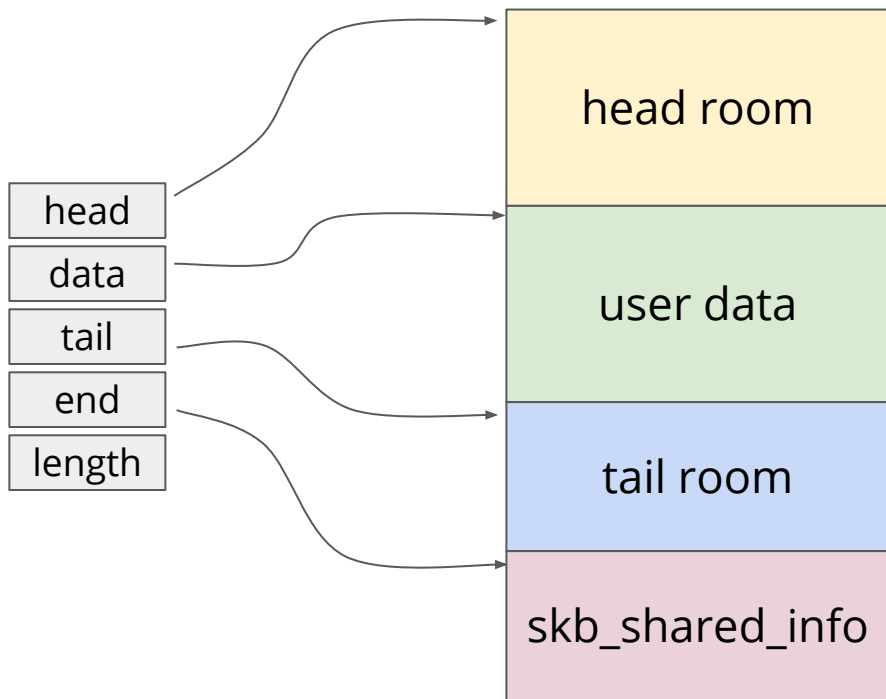
SKB Basic Idea

Logically it contains

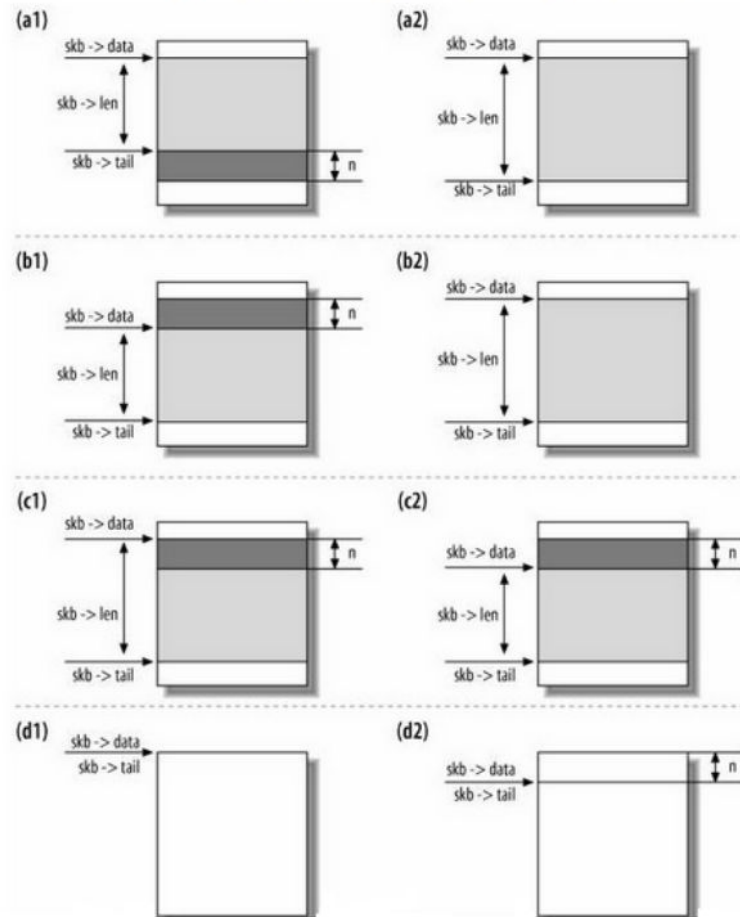
- link list pointers
- which netdev
- which socket
- place for headers / trailers for various protocols
- various Linux specific accounting and reference counting information



SKB Pointers and Operations



Before and after (a) skb_put (b) skb_push (c) skb_pull (d) skb_reserve



Continuing the Data Receiving: main functions

```
/*  
 * Deliver IP Packets to the higher protocol layers.  
 */  
int ip_local_deliver(struct sk_buff *skb)  
{  
    /*  
     * Reassemble IP fragments.  
     */  
    struct net *net = dev_net(skb->dev);
```

It's for us

routing?

Packet needs forwarding

```
/*  
 * IP receive entry point  
 */  
int ip_rcv(struct sk_buff *skb, struct net_device *dev, struct packet_type *pt,  
           struct net_device *orig_dev)  
{  
    struct net *net = dev_net(dev);
```

```
int ip_forward(struct sk_buff *skb)  
{  
    u32 mtu;  
    struct iphdr *iph;    /* Our header */  
    struct rtable *rt;    /* Route we use */  
    struct ip_options *opt = &(IPCB(skb)->opt);  
    struct net *net;  
  
    /* that should never happen */  
    if (skb->pkt_type != PACKET_HOST)  
        goto drop;
```

IP tx path

IP Processing : pushing it to the transport

1. If there is a need for packet assembly
2. If there are networking packet processing rules (netfilter)
 - a. Special hooks
 - b. Programmable code
 - c. Accounting and memory management
3. Lastly, once the whole packet is ready, find the appropriate transport protocol function and call it

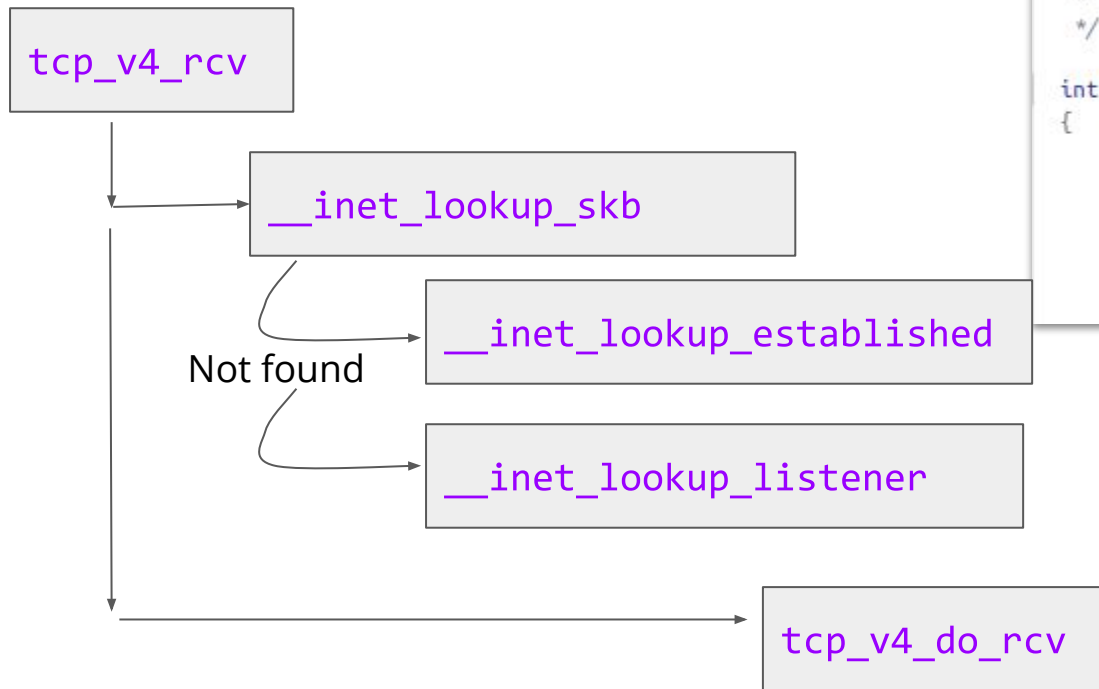
```
/* Build a new IP datagram from all its fragments. */  
static int ip_frag_reasm(struct ipq *qp, struct sk_buff *skb,  
                        struct sk_buff *prev_tail, struct net_device *dev)  
{  
    struct net *net = qp->q.fqdir->net;  
    struct iphdr *iph;  
    void *reassembled_data;  
    int len, err;
```

```
static int ip_local_deliver_finish(struct net *net, struct sock *sk, struct sk_buff *skb)  
{  
    __skb_pull(skb, skb_network_header_len(skb));
```

```
ret = INDIRECT_CALL_2(ipprot->handler, tcp_v4_rcv, udp_rcv,  
                      skb);
```

PS~ we are still in a softirq

Transport Entry Point: `tcp_v4_rcv`



```
/*  
 *      From tcp_input.c  
 */  
  
int tcp_v4_rcv(struct sk_buff *skb)  
{  
    struct net *net = dev_net(skb->dev);  
    struct sk_buff *skb_to_free;  
    int sdif = inet_sdif(skb);  
    int dif = inet_iif(skb);  
    const struct iphdr *iph;  
    const struct tcphdr *th;
```

TCP Packet Processing

```
int tcp_v4_do_rcv(struct sock *sk, struct sk_buff *skb)
{
    struct sock *rsk;

    if (sk->sk_state == TCP_ESTABLISHED) { /* Fast path */
        struct dst_entry *dst = sk->sk_rx_dst;

        sock_rps_save_rxhash(sk, skb);
        sk_mark_napi_id(sk, skb);
        if (dst) {
            if (inet_sk(sk)->rx_dst_ifindex != skb->skb_iif ||
                !dst->ops->check(dst, 0)) {
                dst_release(dst);
                sk->sk_rx_dst = NULL;
            }
        }
        tcp_rcv_established(sk, skb);
        return 0;
    }
}
```

Data processing
in ESTB. state

```
/*
 * TCP receive function for the ESTABLISHED state.
 *
 * It is split into a fast path and a slow path. The fast path is
 * disabled when:
 * - A zero window was announced from us - zero window probing
 *   is only handled properly in the slow path.
 * - Out of order segments arrived.
 * - Urgent data is expected.
 * - There is no buffer space left
 * - Unexpected TCP flags/window values/header lengths are received
 *   (detected by checking the TCP header against pred_flags)
 * - Data is sent in both directions. Fast path only supports pure senders
 *   or pure receivers (this means either the sequence number or the ack
 *   value must stay constant)
 * - Unexpected TCP option.
 *
 * When these conditions are not satisfied it drops into a standard
 * receive procedure patterned after RFC793 to handle all cases.
 * The first three cases are guaranteed by proper pred_flags setting,
 * the rest is checked inline. Fast processing is turned on in
 * tcp_data_queue when everything is OK.
 */
void tcp_rcv_established(struct sock *sk, struct sk_buff *skb)
{
    const struct tcphdr *th = (const struct tcphdr *)skb->data;
    struct tcp_sock *tp = tcp_sk(sk);
    unsigned int len = skb->len;
}
```

State machine processing

```
/*
 * This function implements the receiving procedure of RFC 793 for
 * all states except ESTABLISHED and TIME_WAIT.
 * It's called from both tcp_v4_rcv and tcp_v6_rcv and should be
 * address independent.
 */
int tcp_rcv_state_process(struct sock *sk, struct sk_buff *skb)
{
    struct tcp_sock *tp = tcp_sk(sk);
    struct inet_connection_sock *icsk = inet_csk(sk);
}
```

After the state processing ...

```
/* step 7: process the segment text */  
tcp_data_queue(sk, skb);
```

```
tcp_data_snd_check(sk);  
tcp_ack_snd_check(sk);
```

Check if TX is suspended

Check if ACK needs to be sent

eventually

```
static int __must_check tcp_queue_rcv(struct sock *sk, struct sk_buff *skb,  
                                       bool *fragstolen)  
{  
    int eaten;  
    struct sk_buff *tail = skb_peek_tail(&sk->sk_receive_queue);  
  
    eaten = (tail &&  
            tcp_try_coalesce(sk, tail,  
                            skb, fragstolen)) ? 1 : 0;  
    tcp_rcv_nxt_update(tcp_sk(sk), TCP_SKB_CB(skb)->end_seq);  
    if (!eaten) {  
        __skb_queue_tail(&sk->sk_receive_queue, skb);  
        skb_set_owner_r(skb, sk);  
    }  
    return eaten;  
}
```

- Each socket has a receive queue associated with it so the incoming packets are queue there
- When a user application calls "recv" then this queue is processed and consumed

How does the application side look like?

```
/*
 * This routine copies from a sock struct into the user buffer.
 *
 * Technical note: in 2.3 we work on _locked_ socket, so that
 * tricks with *seq access order and skb->users are not required.
 * Probably, code can be easily improved even more.
 */

int tcp_recvmsg(struct sock *sk, struct msghdr *msg, size_t len, int nonblock,
               int flags, int *addr_len)
{
    struct tcp_sock *tp = tcp_sk(sk);
    int copied = 0;
    u32 peek_seq;
    u32 *seq;
    unsigned long used;
    int err, inq;
    int target;           /* Read at least this many bytes */
    long timeo;
    struct sk_buff *skb, *last;
    u32 urg_hole = 0;
    struct scm_timestamping_internal tss;
```

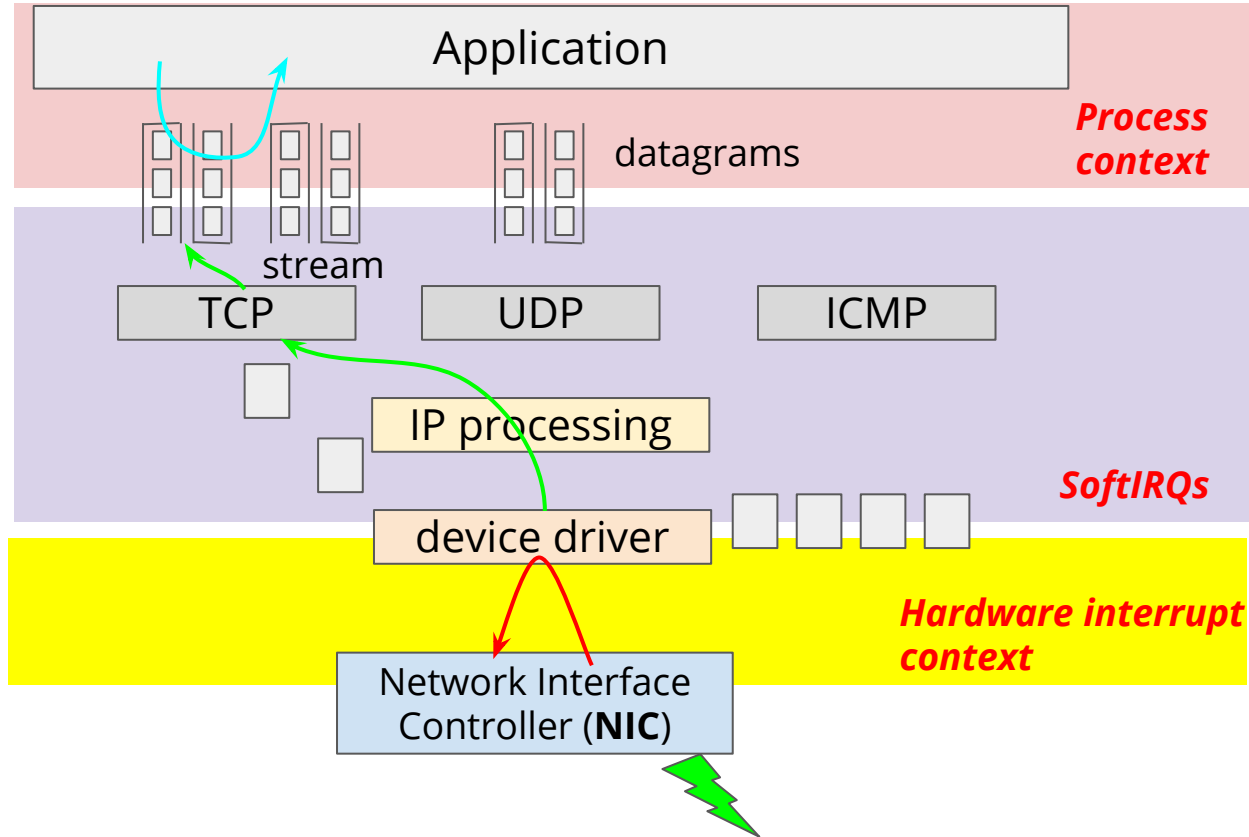
recv(...)



- If there is enough data available on the socket receive queue then copy the data out
- Otherwise block and wait until more data arrives
 - Also *non-blocking* modes possible

ps~ this is NOT in a softirq

What are the key components that run



*Shared data structures
must be protected properly
from concurrent execution*



In the ANP project (at least 3 threads)

```
void *netdev_rx_loop()
{
    int ret;
    while (!stop) {
        // The max size of ethernet packet
        // https://searchnetworking.techtarget.com/definition/Ethernet-frame
        struct subuff *sub = malloc(sizeof(subuff));
        ret = tdev_read((char *)sub, sizeof(subuff));
        if (ret < 0) {
            printf("Error in read: %d\n", ret);
            free_sub(sub);
            return NULL;
        }
        // whatever we have received, process it
        process_packet(sub);
    }
    return NULL;
}
```

**TCP client
send/recv**



TAP device

```
void *timers_start()
{
    //10 millisecond timer loop
    while (1) {
        if (usleep(10000) != 0) {
            perror("Timer usleep");
        }

        pthread_rwlock_wrlock(&rwlock);
        tick += 10;
        pthread_rwlock_unlock(&rwlock);
        timers_tick();
    }
}
```

Timer thread

TCP Sending Path

Sending path is (relatively) ~~simpler~~ less complex than the receiving path, **why?**

Receiving path is very asynchronous, *what does this mean? Lots of events happening that one cannot schedule properly on the time of their choosing:*

- Reception of a packet
- Interrupts
- Timeouts
- Application calling receive

Often in network benchmarking you will see the receiving side becoming CPU bottleneck before the sending side.

That means we need to be ready at any time to process events

In comparison to this, sending side is considered synchronous, that means we can control what happens when on the sending path

- There are no unexpected events

When does send processing starts?

```
int tcp_sendmsg(struct sock *sk, struct msghdr *msg, size_t size)
{
    int ret;

    lock_sock(sk);
    ret = tcp_sendmsg_locked(sk, msg, size);
    release_sock(sk);

    return ret;
}
```

```
static inline struct sk_buff *tcp_write_queue_tail(const struct sock *sk)
{
    return skb_peek_tail(&sk->sk_write_queue);
}
```

socket write queue

```
mss_now = tcp_send_mss(sk, &size_goal, flags);

err = -EPIPE;
if (sk->sk_err || (sk->sk_shutdown & SEND_SHUTDOWN))
    goto do_error;

while (msg_data_left(msg)) {
    int copy = 0;

    skb = tcp_write_queue_tail(sk);
    if (skb)
        copy = size_goal - skb->len;
}
```

```
/* Where to copy to? */
if (skb_availroom(skb) > 0 && !zc) {
    /* We have some space in skb head. Superb! */
    copy = min_t(int, copy, skb_availroom(skb));
    err = skb_add_data_nocache(sk, skb, &msg->msg_iter, copy);
    if (err)
        goto do_fault;
} else if (!zc) {
    bool merge = true;
    int i = skb_shinfo(skb)->nr_frags;
```

Data is copied into the kernel space from user space

After Copying ...

```
/* Send_single_skb sitting at the send head. This function requires
 * true push pending frames to setup probe timer etc.
 */
void tcp_push_one(struct sock *sk, unsigned int mss_now)
{
    struct sk_buff *skb = tcp_send_head(sk);

    BUG_ON(!skb || skb->len < mss_now);

    tcp_write_xmit(sk, mss_now, TCP_NAGLE_PUSH, 1, sk->sk_allocation);
}
```

```
/* This routine writes packets to the network. It advances the
 * send_head. This happens as incoming acks open up the remote
 * window for us.
 *
 * LARGESEND note: !tcp_urg_mode is overkill, only frames between
 * snd_up-64k-mss .. snd_up cannot be large. However, taking into
 * account rare use of URG, this is not a big flaw.
 *
 * Send at most one packet when push_one > 0. Temporarily ignore
 * cwnd limit to force at most one packet out when push_one == 2.
 *
 * Returns true, if no segments are in flight and we have queued segments,
 * but cannot send anything now because of SWS or another problem.
 */
static bool tcp_write_xmit(struct sock *sk, unsigned int mss_now, int nonagle,
                          int push_one, gfp_t gfp)
{
    struct tcp_sock *tp = tcp_sk(sk);
    struct sk_buff *skb;
    unsigned int tso_segs, sent_pkts;
    int cwnd_quota;
    int result;
    bool is_cwnd_limited = false, is_rwnd_limited = false;
```

Checks all TCP parameters, ACKs,
window size

Building an actual TCP header
And pushing to the IP layer

```
/* This routine actually transmits TCP packets queued in by
 * tcp_do_sendmsg(). This is used by both the initial
 * transmission and possible later retransmissions.
 * All SKB's seen here are completely headerless. It is our
 * job to build the TCP header, and pass the packet down to
 * IP so it can do the same plus pass the packet off to the
 * device.
 *
 * We are working here with either a clone of the original
 * SKB, or a fresh unique copy made by the retransmit engine.
 */
static int tcp_transmit_skb(struct sock *sk, struct sk_buff *skb,
                           int clone_it, gfp_t gfp_mask, u32 rcv_nxt)
{
    const struct inet_connection_sock *icsk = inet_csk(sk);
    struct inet_sock *inet;
    struct tcp_sock *tp;
    struct tcp_skb_cb *tcb;
    struct tcp_out_options opts;
```

ip_queue_xmit()

In the IP layer

Build the IP packet here

```
/* Note: skb->sk can be different from sk, in case of tunnels */
int __ip_queue_xmit(struct sock *sk, struct sk_buff *skb, struct flowi *fl,
                   __u8 tos)
{
    struct inet_sock *inet = inet_sk(sk);
    struct net *net = sock_net(sk);
    struct ip_options_rcu *inet_opt;
    struct flowi4 *fl4;
    struct rtable *rt;
    struct iphdr *iph;
    int res;

    /* Skip all of this if the packet is already routed,
     * f.e. by something like SCTP.
     */
    rcu_read_lock();
    inet_opt = rcu_dereference(inet->inet_opt);
    fl4 = &fl->u.ip4;
```

Route resolution

```
int ip_output(struct net *net, struct sock *sk, struct sk_buff *skb)
{
    struct net_device *dev = skb_dst(skb)->dev, *indev = skb->dev;

    IP_UPD_PO_STATS(net, IPSTATS_MIB_OUT, skb->len);

    skb->dev = dev;
    skb->protocol = htons(ETH_P_IP);
```

```
static int __ip_finish_output(struct net *net, struct sock *sk, struct sk_buff *skb)
{
    unsigned int mtu;

    #if defined(CONFIG_NETFILTER) && defined(CONFIG_XFRM)
    /* Policy lookup after SNAT yielded a new policy */
    if (skb_dst(skb)->xfrm) {
        IPCB(skb)->flags |= IPSKB_REROUTED;
        return dst_output(net, sk, skb);
    }
```

Back into netdev structure and functions...

```
static int __dev_queue_xmit(struct sk_buff *skb, struct net_device *sb_dev)
{
    struct net_device *dev = skb->dev;
    struct netdev_queue *txq;
    struct Qdisc *q;
    int rc = -ENOMEM;
    bool again = false;

    skb_reset_mac_header(skb);

    if (unlikely(skb_shinfo(skb)->tx_flags & SKBTX_SCHED_TSTAMP))
        __skb_tstamp_tx(skb, NULL, skb->sk, SCM_TSTAMP_SCHED);

    /* Disable soft irqs for various locks below. Also
     * stops preemption for RCU.
     */
```

```
struct sk_buff *dev_hard_start_xmit(struct sk_buff *first, struct net_device *dev,
                                   struct netdev_queue *txq, int *ret)
{
    struct sk_buff *skb = first;
    int rc = NETDEV_TX_OK;

    while (skb) {
        struct sk_buff *next = skb->next;

        skb_mark_not_on_list(skb);
        rc = xmit_one(skb, dev, txq, next != NULL);
        if (unlikely(!dev_xmit_complete(rc))) {
            skb->next = next;
            goto out;
        }
    }
}
```

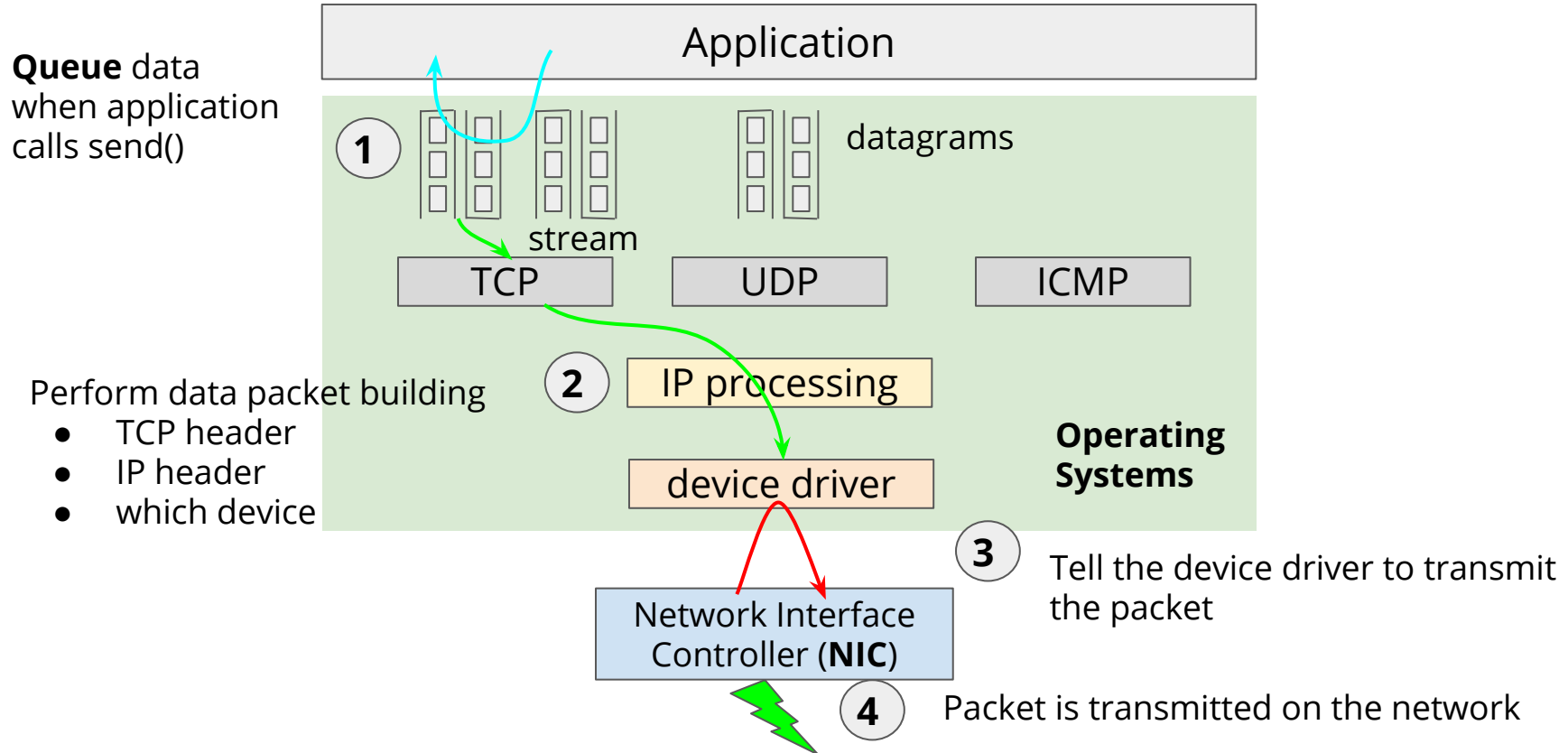
Various decision regarding packet scheduling, quality of service, packing packet rates, etc. are made here

A different kernel subsystem: **qdiscs**

<https://www.linuxjournal.com/content/queuing-linux-network-stack>

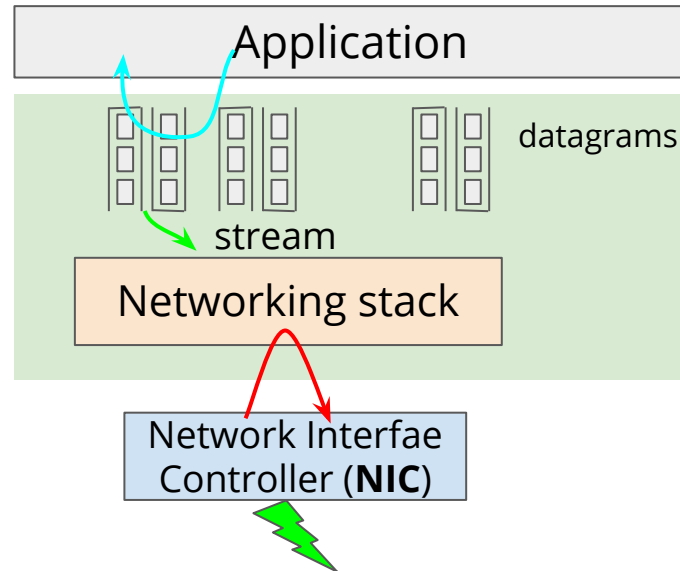
Actual transmission can run in softirq

A Packet's Journey - (simplified) Sending Path



Zero-copy Transmission

Question: how many time data is copied when doing a packet transmission?



Zero-copy Transmission

Question: how many time data is copied when doing a packet transmission?

Answer: 1 time, when copying out from the user buffer into the kernel buffer

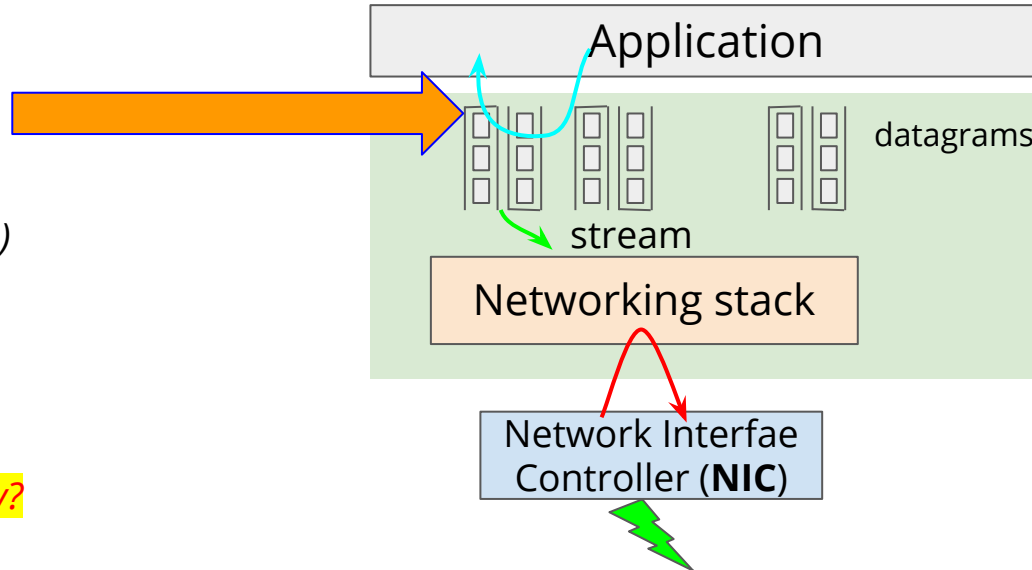
- DMA is not counted as a copy (NIC is doing it, not the CPU), 2x memory crossing

Copy is happening here

- From: user buffer
- To: kernel buffer (in SKB)

sk->sk_write_queue

Why copy is necessary?



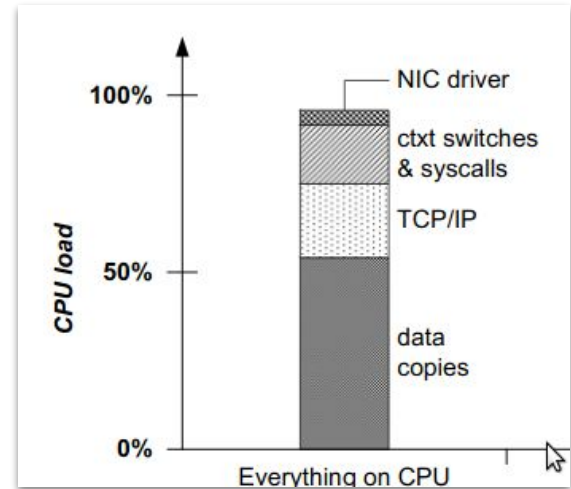
Why do zero-copy?

Can we do better? Before we answer that, why copying is necessary?

- Kernel cannot control when the DMA happens
- User process can be (de)scheduled at any time, hence, memory mappings are not valid
- Kernel need access to data in case of retransmission
- Kernel need to respect restrictions on memory alignment, location, and sizes (that a NIC can DMA or I/O to)

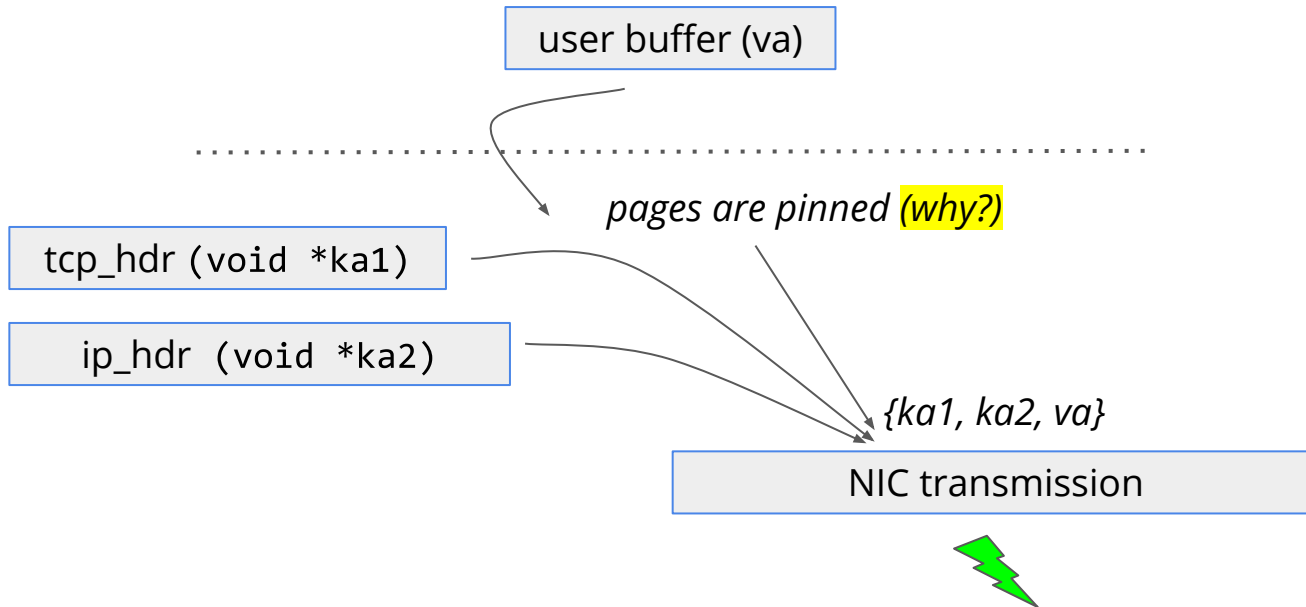
What would it bring?

- Free CPU from doing data copy
- Better performance
- Free CPU cycles for execution of application
- Better energy efficiency



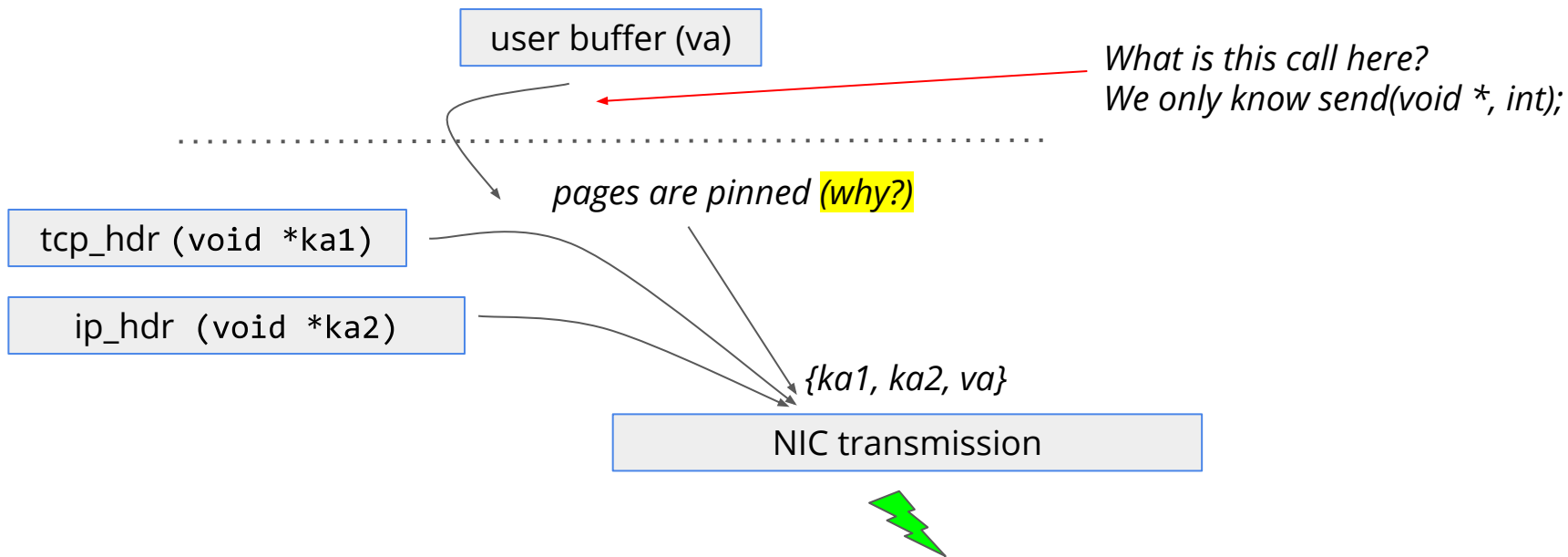
How it might look like?

Assuming that the NIC has architectural features to do arbitrary DMA



How it might look like?

Assuming that the NIC has architectural features to do arbitrary DMA



MSG_ZEROCOPY: New Linux feature (only for TCP)

```
if (setsockopt(fd, SOL_SOCKET, SO_ZEROCOPY, &one, sizeof(one)))  
    error(1, errno, "setsockopt zerocopy");
```

register your intent (legacy-proof)

```
ret = send(fd, buf, sizeof(buf), MSG_ZEROCOPY);
```

pass the additional flag: MSG_ZEROCOPY

```
pfd.fd = fd;  
pfd.events = 0;  
if (poll(&pfd, 1, -1) != 1 || pfd.revents & POLLERR == 0)  
    error(1, errno, "poll");  
  
ret = recvmsg(fd, &msg, MSG_ERRQUEUE);  
if (ret == -1)  
    error(1, errno, "recvmsg");  
  
read_notification(msg);
```

Wait until it is safe to reuse the buffer again

why?



- https://www.kernel.org/doc/html/latest/networking/msg_zerocopy.html
- <https://netdevconf.info/2.1/session.html?debruijn>

How much does it save us?

```
NETPERF=./netperf -t TCP_STREAM -H $host -T 2 -l 30 -- -m $size
```

```
perf stat -e cycles $NETPERF
```

```
perf stat -C 2,3 -a -e cycles $NETPERF
```

Application cycle

--process cycles--

| | std | zc | % |
|------|--------|--------|----|
| 4K | 27,609 | 11,217 | 41 |
| 16K | 21,370 | 3,823 | 18 |
| 64K | 20,557 | 2,312 | 11 |
| 256K | 21,110 | 2,134 | 10 |
| 1M | 20,987 | 1,610 | 8 |

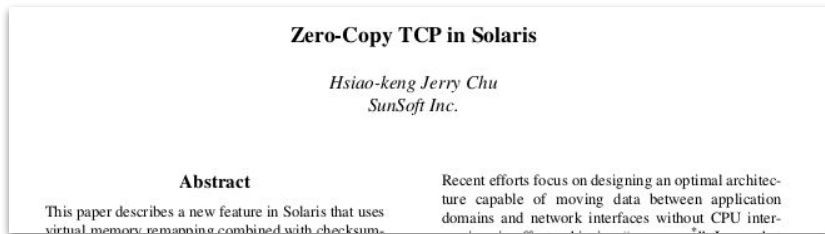
system-wide

----cpu cycles----

| | std | zc | % |
|------|--------|--------|----|
| 4K | 49,217 | 39,175 | 79 |
| 16K | 43,540 | 29,213 | 67 |
| 64K | 42,189 | 26,910 | 64 |
| 256K | 43,006 | 27,104 | 63 |
| 1M | 42,759 | 25,931 | 61 |

Long History of Zero Copy Stacks

- There is a long history of developing zero-copy stacks (1980s)
 - Often limitations were limited due to gains in the CPU performance due to Moore's law
 - Many architectural pitfalls:
<https://www.usenix.org/conference/usenix-1996-annual-technical-conference/zero-copy-tcp-solaris> (very good reading)
- In Linux today, there are more calls
 - sendfile, splice, vmsplice (check their man pages, to transfer data between fd and pipes)



Can you implement a zero-copy receiving stack?

We will revisit the idea of zero-copy stack later in the course

Types of Optimization(s) or Overhead(s)

Often in paper/research reports you will see when talking about overheads or optimization, there are two classes of operations

1. **Per-packet operations**
2. **Per-byte operations**

What are these?

Overhead(s)

Operations that needs to be done on **per packet basis**

Examples:

1. Generation of TCP / IP packets
2. Allocation of SKB structure
3. TCP state machine transition
4. ACK generation
5. Queue management

Cost increases with the number of packets

Operations that needs to be done on **per-byte** basis in any packet

Examples:

1. Data copies
2. Checksum generation
3. DMA
4. IPSec (encryption/decryption)

Cost increases with the number of bytes

Classify these optimization for per-byte or per-packet

- Jumbo packets
- TCP segmentation offloading (TSO)
- Checksum offloading
- Large Receive Offload (LRO)
- Interrupt coalescing
- Scatter-gather I/O capabilities
- Zero-copy stack

*Think: do they change the **number of packet** or **number of bytes** that a CPU need to process*

Key Message Here is

Networking does not happen in isolation

- Memory management: SKB, rx/tx queues, user buffers
- Scheduling : softirq, kthreads, user threads
- Device management : NAPI, ethtool, MTU management
- Architectural implications : DMA, alignment, system calls, interrupts
- ...and everything happening inside an operating system

Sending and receiving data have many parameters that can be set and optimized for application's needs, check the man pages

```
atr@atr:~$ man 7 tcp
```

```
atr@atr:~$ man 7 ip
```

```
atr@atr:~$ man 7 socket
```

Recap: What You Should Know from this Lecture

1. A general idea about Linux networking stack internals
 - a. How is the source code organized
2. What is a SoftIRQ, top-half, and bottom-half processing
 - a. What you can and can not in them
3. What is a NAPI, what is it used for
4. What is a SKB
5. What happens when a user application calls : send(), recv()
6. What is a zero/one copy stack? What is it good for?

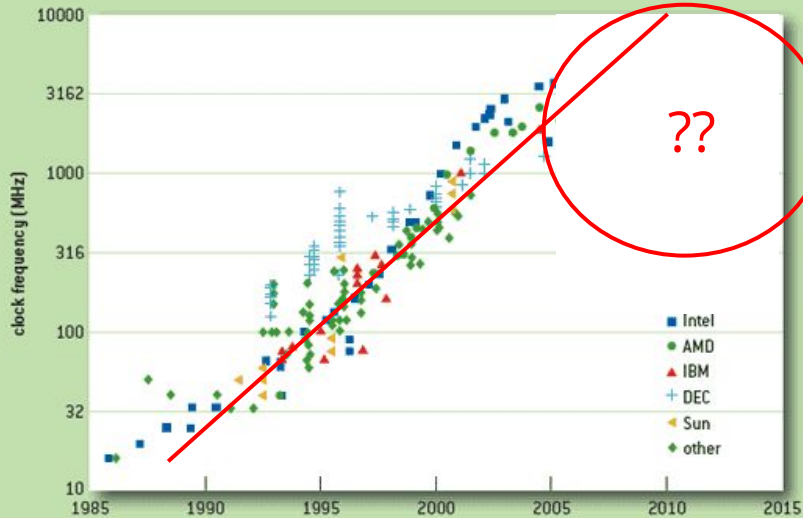
Next week we will see what challenges this basic stack has to deal with in presence of high performance networks like 100 Gbps

Next Lecture: Multicore scalability



FIGURE 7

Processor Frequency Scaling Over Time



But then something happened here, and all our dreams of 10 GHz CPU were shattered ;)



Useful links and references (some briefly outdated)

1. The linux networking architecture, <https://www.slideshare.net/hugolu/the-linux-networking-architecture>
2. TCP Implementation in Linux: A Brief Tutorial, <https://sn0rt.github.io/media/paper/TCPlinux.pdf>
3. Path of a packet in the Linux kernel stack,
https://www.cs.dartmouth.edu/~sergey/netreads/path-of-packet/Network_stack.pdf
4. Understanding Linux Network Internals, Book by Christian Benvenuti,
5. Linux kernel documentation: <https://linux-kernel-labs.github.io/refs/heads/master/labs/networking.html>