

Linux Network Stack

接收流程

前置准备

在开始收包之前，也就是OS启动的时候，Linux要做许多的准备工作：

1. 创建ksoftirqd线程，为它设置好它自己的线程函数，用来处理软中断
2. 协议栈注册，linux要实现许多协议，比如arp，icmp，ip，udp，tcp，每一个协议都会将自己的处理函数注册一下，方便包来了迅速找到对应的处理函数
3. 网卡驱动初始化，每个驱动都有一个初始化函数，内核会让驱动也初始化一下。在这个初始化过程中，把自己的DMA准备好，把NAPI的poll函数地址告诉内核
4. 启动网卡，分配RX，TX队列，注册中断对应的处理函数

网卡处理

数据到来了以后，第一个迎接它的是网卡：

1. 网卡将数据帧DMA到内存的RingBuffer中，然后向CPU发起中断通知
2. CPU响应中断请求，调用网卡启动时注册的中断处理函数
3. 中断处理函数几乎没干啥，就发起了软中断请求
4. 内核线程ksoftirqd线程发现有软中断请求到来，先关闭硬中断
5. ksoftirqd线程开始调用驱动的poll函数收包
6. poll函数将收到的包送到协议栈注册的ip_rcv函数中
7. ip_rcv函数再讲包送到udp_rcv函数中（对于tcp包就送到tcp_rcv）

接收流程

1. 网络包进到网卡，网卡驱动校验MAC，看是否扔掉，取决是否是混杂 promiscuous mode
2. 网卡在启动时会申请一个接收ring buffer，其条目都会指向一个skb的内存。
3. DMA完成数据报文从网卡硬件到内存到拷贝
4. 网卡发送一个中断通知CPU。
5. CPU执行网卡驱动注册的中断处理函数，中断处理函数只做一些必要的工作，如读取硬件状态等，并把当前该网卡挂在NAPI的链表中；
6. Driver “触发” soft IRQ(NET_RX_SOFTIRQ (其实也就是设置对应软中断的标志位)

7. CPU中断处理函数返回后，会检查是否有软中断需要执行。因第6步设置了NET_RX_SOFTIRQ，则执行报文接收软中断。

8. 在NET_RX_SOFTIRQ软中断中，执行NAPI操作，回调第5步挂载的驱动poll函数。

9. 驱动会对interface进行poll操作，检查网卡是否有接收完毕的数据报文。

10. 将网卡中已经接收完毕的数据报文取出，继续在软中断进行后续处理。注：驱动对interface执行poll操作时，会尝试循环检查网卡是否有接收完毕的报文，直到系统设置的net.core.netdev_budget上限(默认300)，或者已经就绪报文。

11. net_rx_action

12. 内核分配 sk_buff 内存

13. 内核填充 metadata: 协议等，移除 ethernet 包头信息

14. 将skb 传送给内核协议栈 netif_receive_skb

15. `__netif_receive_skb_core`：将数据送到抓包点 (tap) 或协议层(i.e. tcpdump)// 出抓包点：dev_queue_xmit_nit

16. 进入到由 netdev_max_backlog 控制的qdisc

17. 开始 ip_rcv 处理流程，主要处理ip协议包头相关信息

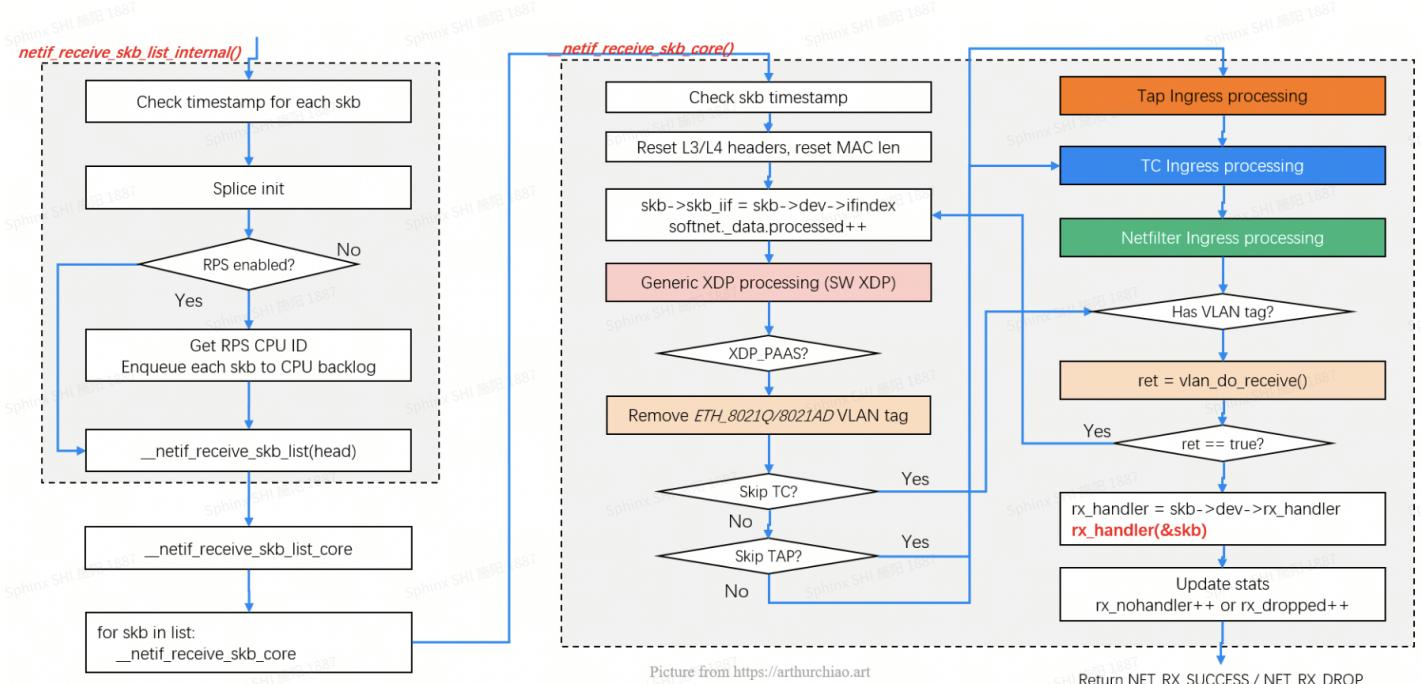
18. 调用内核 netfilter 框架(iptables PREROUTING)

19. 进入L4 protocol `tcp_v4_rcv`

20. 找到对应的socket

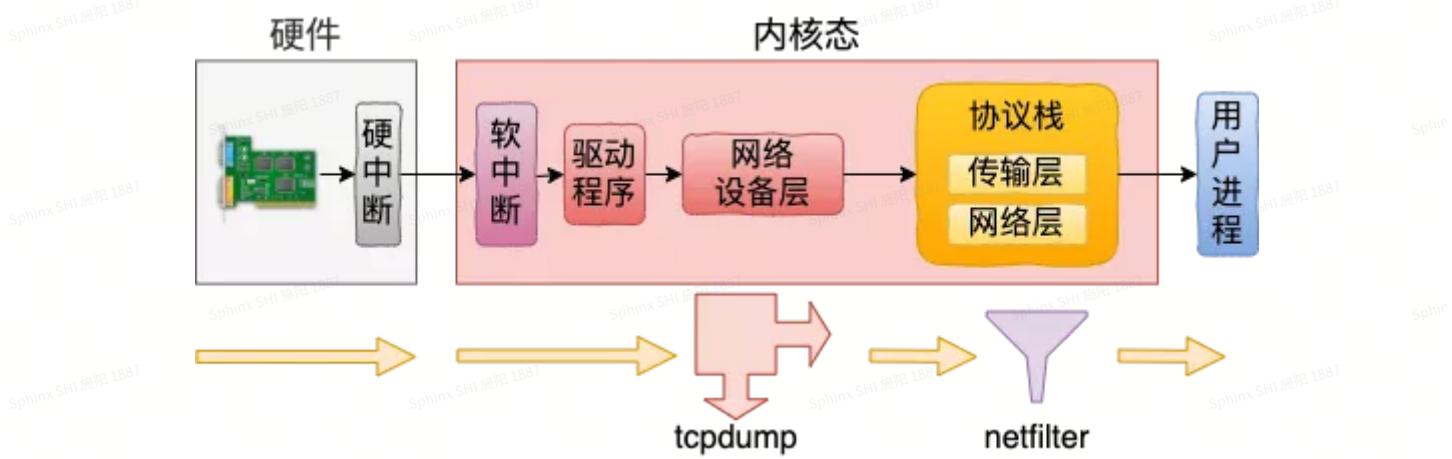
21. 根据 `tcp_rmem` 进入接收缓冲队列

22. 内核将数据送给接收的应用



扩展

TAP 处理点就是 `tcpdump` 抓包、流量过滤。



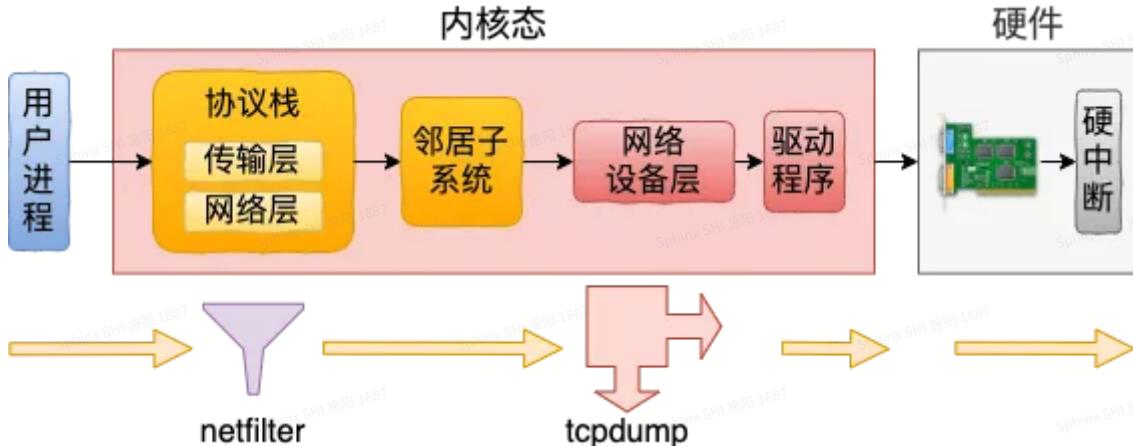
注意：netfilter 或 iptables 规则都是在软中断上下文中执行的，数量很多或规则很复杂时会导致网络延迟。

软中断：可以把软中断系统想象成一系列内核线程（每个 CPU 一个），这些线程执行针对不同事件注册的处理函数（handler）。如果你用过 `top` 命令，可能会注意到 `ksoftirqd/0` 这个内核线程，其表示这个软中断线程跑在 CPU 0 上。

硬中断发生在哪一个核上，它发出的软中断就由哪个核来处理。可以通过加大网卡队列数，这样硬中断工作、软中断工作都会有更多的核心参与进来。

`__napi_schedule` 干两件事情，一件事情是把 `struct napi_struct` 挂到 `struct softnet_data` 上，注意 `softnet_data` 是一个 per cpu 变量，换句话说，软中断结构是挂在触发硬中断的同一个 CPU 上；另一件事情是调用 `_raise_softirq_irqoff` 把 `irq_stat` 的 `__softirq_pending` 字段置位，`irq_stat` 也是个 per cpu 变量，表示当前这个cpu上有软中断待处理。

从上图可以看到 `tcpdump` 在协议栈之前，也就是 `netfilter` 过滤规则对 `tcpdump` 无效，发包则是反过来：



软件工程视角

Data Receiving

Now, let's take a look at how data is received. Data receiving is a procedure for how the network stack handles a packet coming in. **Figure 3** shows how the network stack handles a packet received.

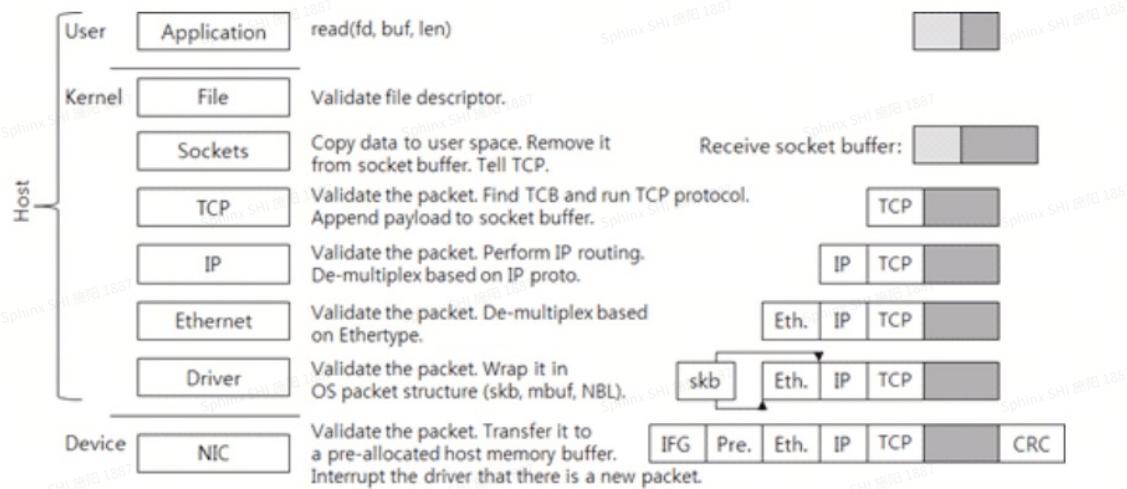
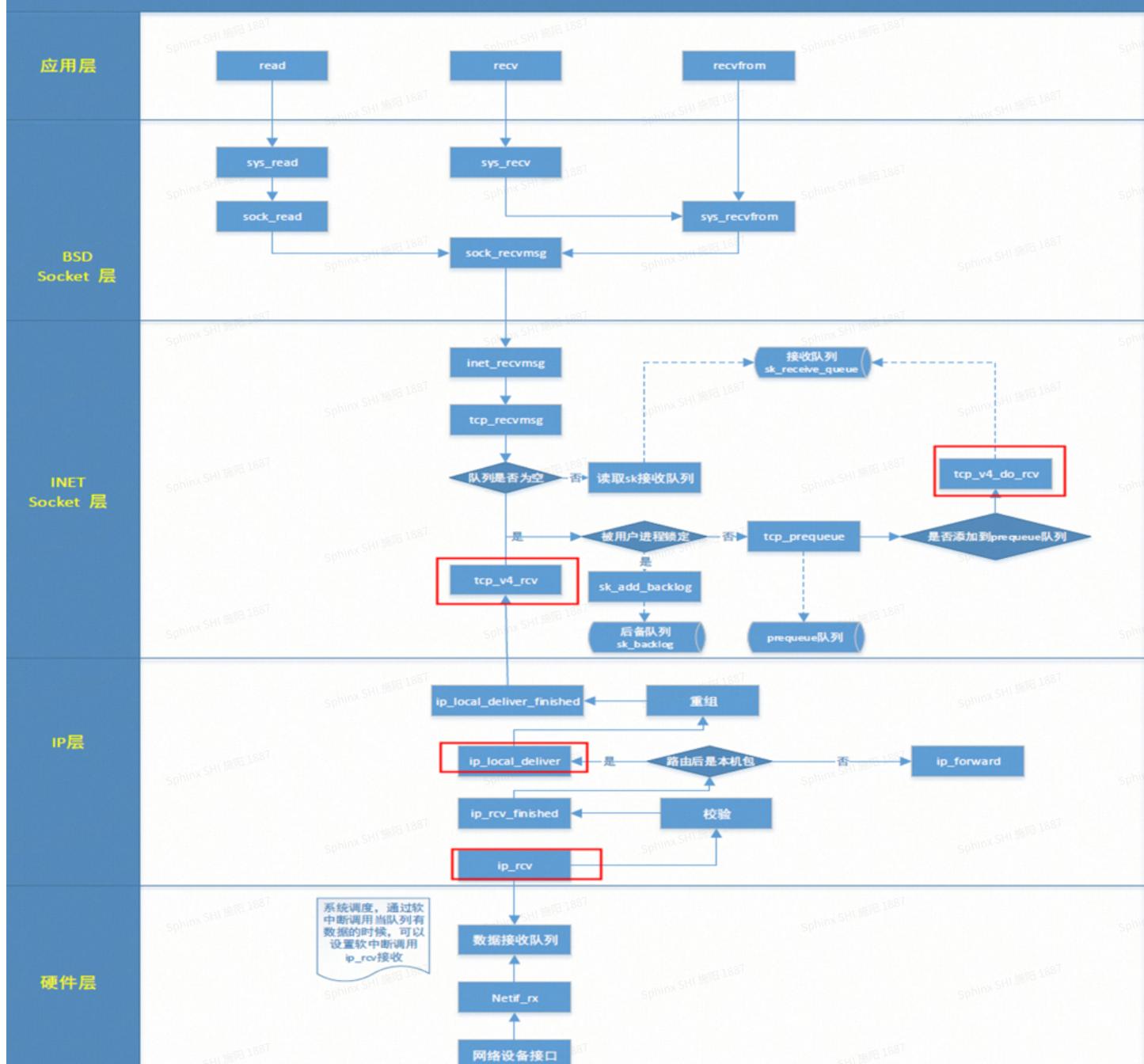


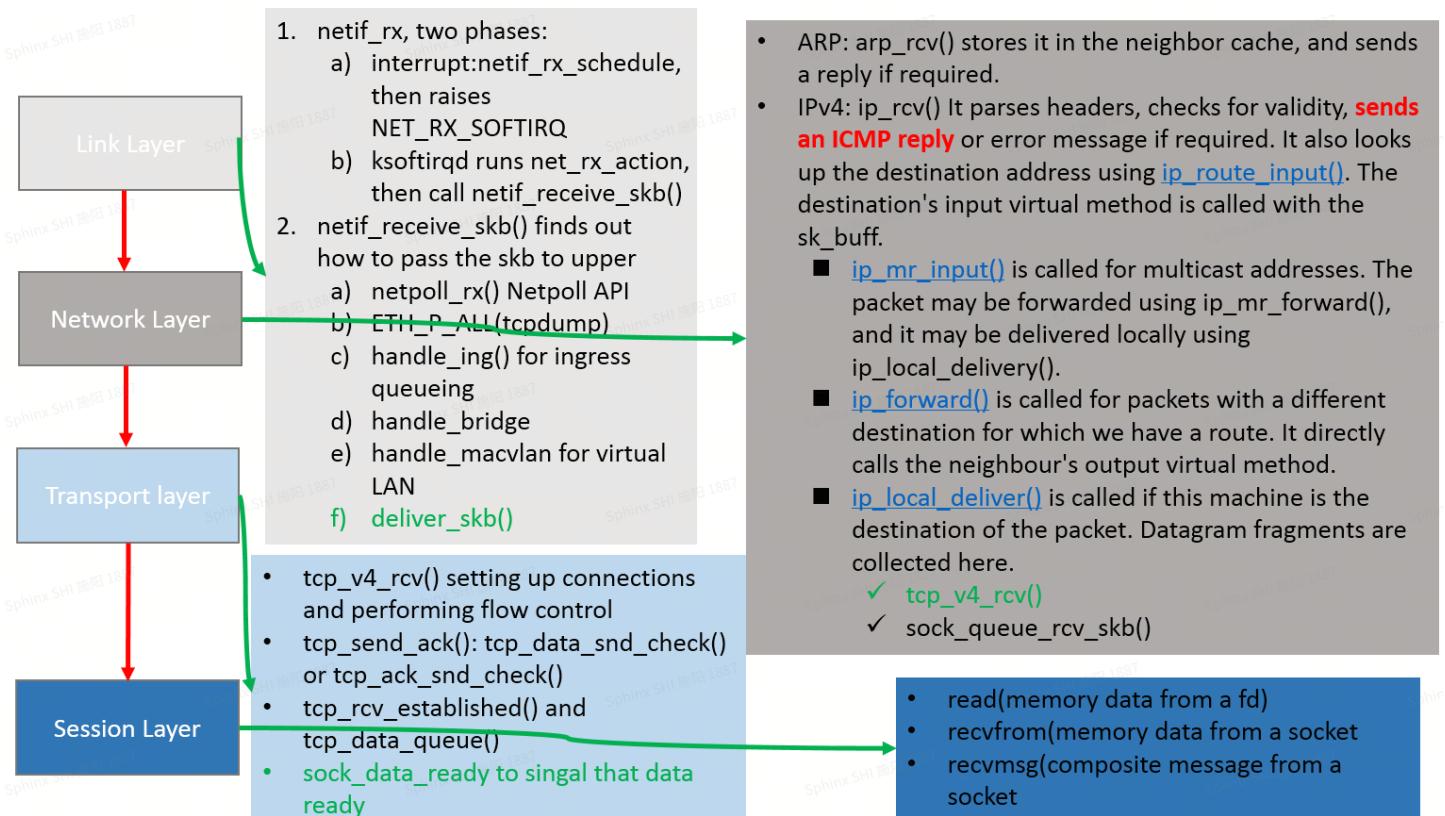
Figure 3: Operation Process by Each Layer of TCP/IP Network Stack for Handling Data Received.

Linux network 视角

Linux内核网络数据接收流程图



TCP/IP 视角



硬件加速: DMA

DMA是一个硬件逻辑，数据传输到系统物理内存的过程中，全程不需要CPU的干预，除了占用总线之外(期间CPU不能使用总线)，没有任何额外开销。

As following the packet transmission procedure shown in the following **Figure 8**, you will see how the ring is used.

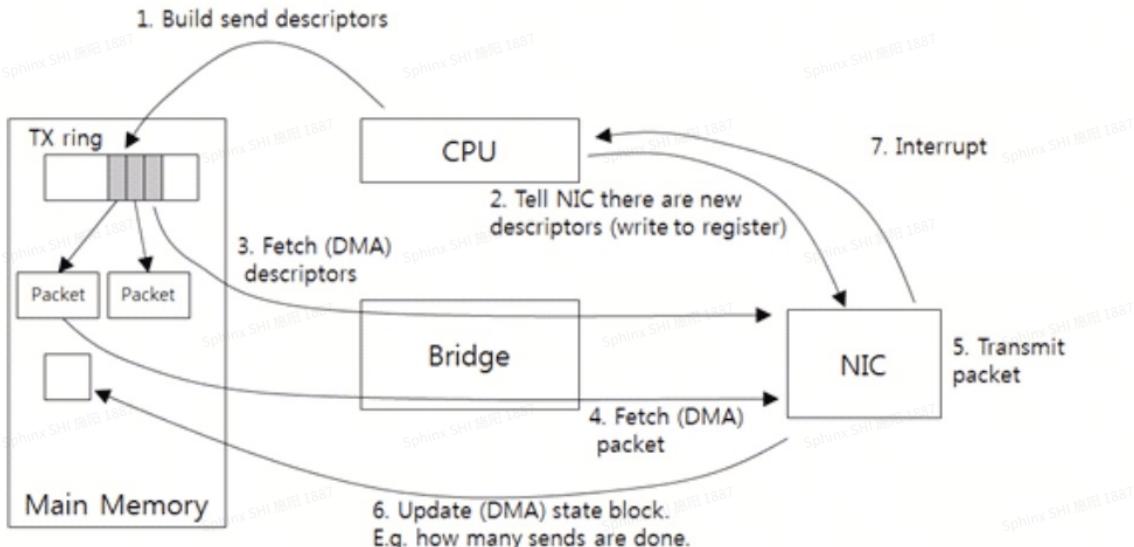
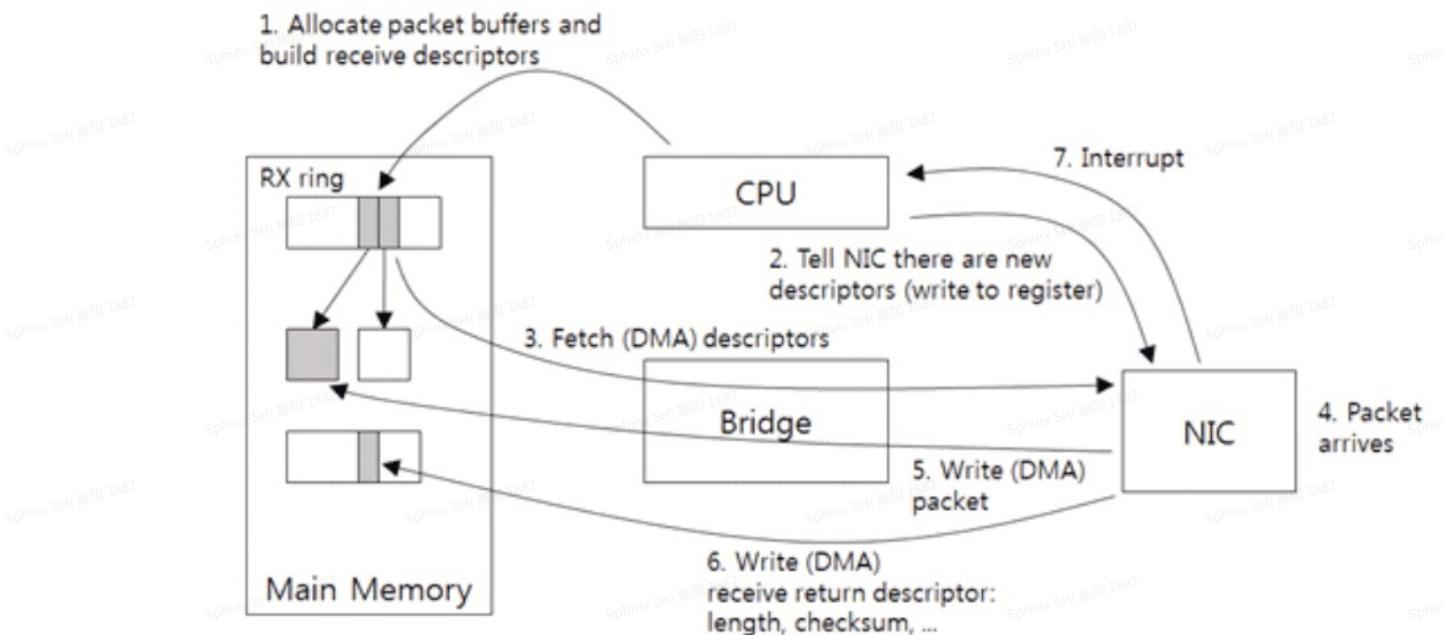
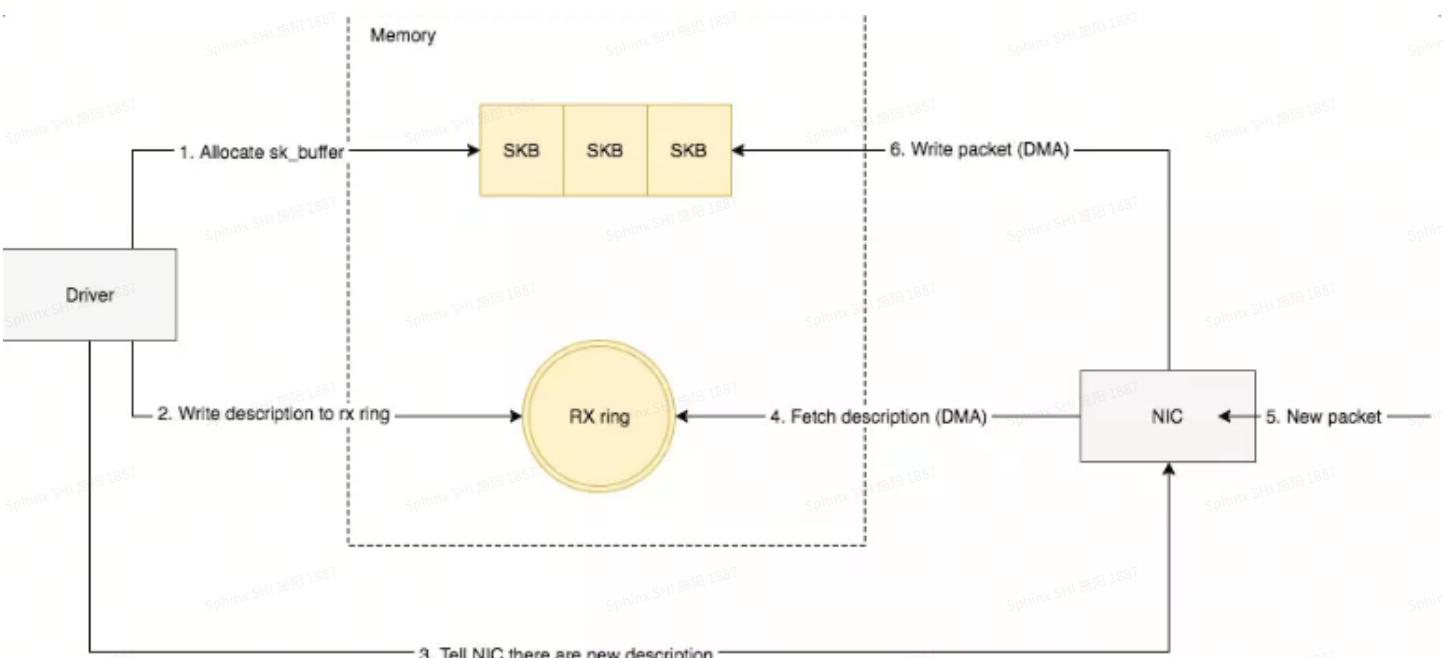


Figure 8: Driver-NIC Communication: How to Transmit Packet.

In the following Figure 9, you will see the procedure of receiving packets.

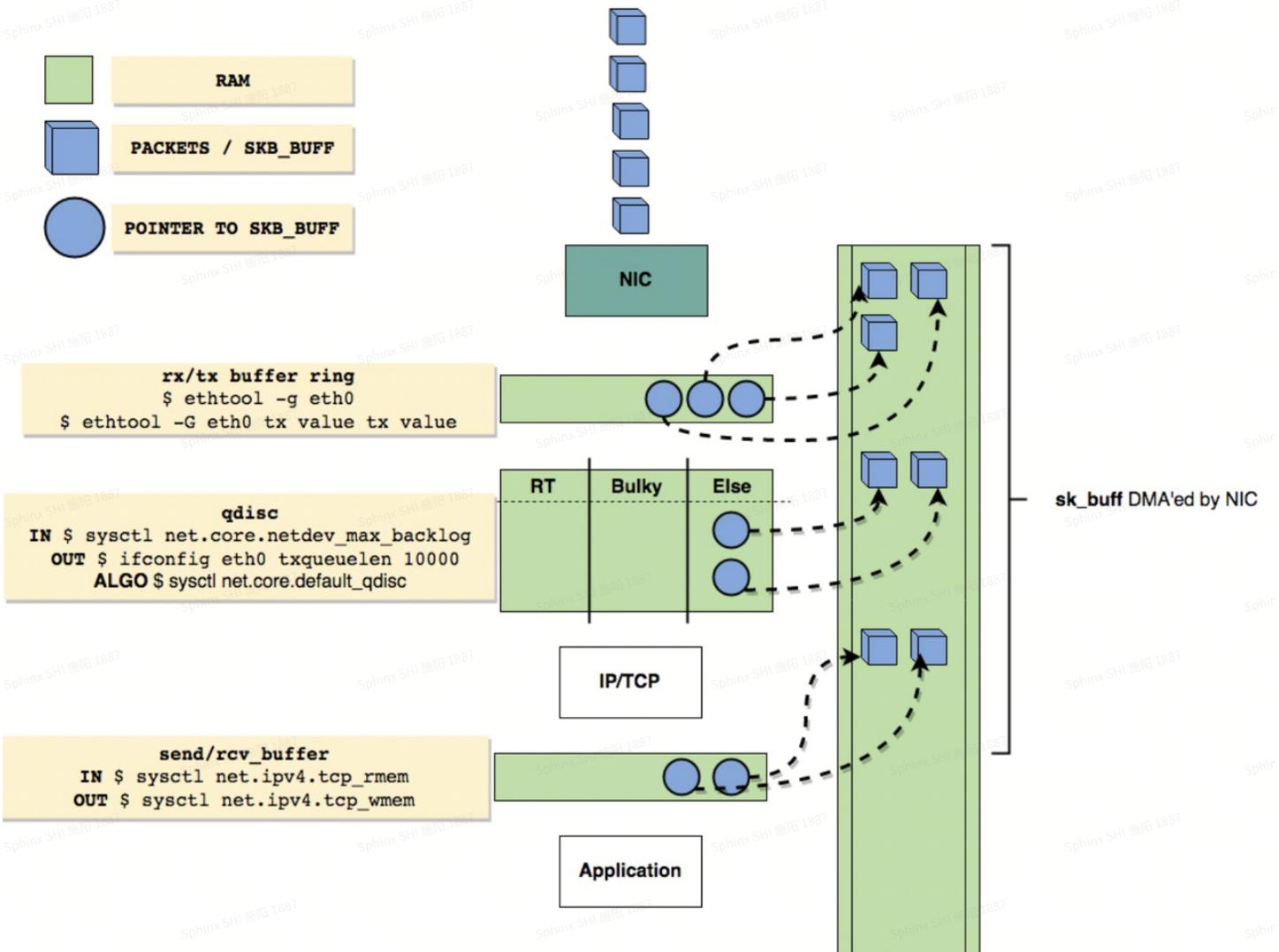


<http://www.cubrid.org/blog/dev-platform/understanding-tcp-ip-network-stack/>



1. 驱动在内存中分配一片缓冲区用来接收数据包，叫做sk_buffer;
2. 将上述缓冲区的地址和大小（即接收描述符），加入到rx ring buffer。描述符中的缓冲区地址是DMA使用的物理地址;
3. 驱动通知网卡有一个新的描述符;
4. 网卡从rx ring buffer中取出描述符，从而获知缓冲区的地址和大小;
5. 网卡收到新的数据包;
6. 网卡将新数据包通过DMA直接写到sk_buffer中。

Linux kernel queues



实践

使用perf来看下实际处理链条：

代码块

```
1 sudo mount -o remount,mode=755 /sys/kernel/tracing/ # 挂载内核 tracing
2 sudo perf trace --no-syscalls --event 'net:*' ping baidu.com -c1 # trace ping
```

output

代码块

```
1      0.000 ping/2736196 net:net_dev_queue(skbaddr: 0xfffff9973d9afb000, len:
80, name: "lo")
2      0.010 ping/2736196 net:net_dev_start_xmit(name: "lo", skbaddr:
0xfffff9973d9afb000, protocol: 2048, ip_summed: 3, len: 80, network_offset: 14,
transport_offset_valid: 1, transport_offset: 34)
3      0.014 ping/2736196 net:netif_rx_entry(name: "lo", napi_id: 27, skbaddr:
0xfffff9973d9afb000, protocol: 2048, ip_summed: 3, hash: 1983044234, l4_hash:
```

```
1, len: 66, truesize: 768, mac_header_valid: 1, mac_header: 4294967282)
 4      0.017 ping/2736196 net:netif_rx(skbaddr: 0xfffff9973d9afb000, len: 66,
    name: "lo")
 5      0.020 ping/2736196 net:netif_rx_exit(skbaddr: 0xfffff9973d9afb000, len:
    66, name: "lo")
 6      0.023 ping/2736196 net:net_dev_xmit(skbaddr: 0xfffff9973d9afb000, len: 80,
    name: "lo")
 7      0.027 ping/2736196 net:netif_receive_skb(skbaddr: 0xfffff9973d9afb000,
    len: 66, name: "lo")
 8      0.037 ping/2736196 net:net_dev_queue(skbaddr: 0xfffff9973d9afb300, len:
    80, name: "lo")
 9      0.040 ping/2736196 net:net_dev_start_xmit(name: "lo", skbaddr:
    0xfffff9973d9afb300, protocol: 2048, ip_summed: 3, len: 80, network_offset: 14,
    transport_offset_valid: 1, transport_offset: 34)
10      0.043 ping/2736196 net:netif_rx_entry(name: "lo", napi_id: 27, skbaddr:
    0xfffff9973d9afb300, protocol: 2048, ip_summed: 3, hash: 1983044234, l4_hash:
    1, len: 66, truesize: 768, mac_header_valid: 1, mac_header: 4294967282)
11      0.045 ping/2736196 net:netif_rx(skbaddr: 0xfffff9973d9afb300, len: 66,
    name: "lo")
12      0.048 ping/2736196 net:netif_rx_exit(skbaddr: 0xfffff9973d9afb300, len:
    66, name: "lo")
13      0.051 ping/2736196 net:net_dev_xmit(skbaddr: 0xfffff9973d9afb300, len: 80,
    name: "lo")
14      0.054 ping/2736196 net:netif_receive_skb(skbaddr: 0xfffff9973d9afb300,
    len: 66, name: "lo")
15 PING baidu.com (110.242.68.66) 56(84) bytes of data.
16      7.650 ping/2736196 net:net_dev_queue(skbaddr: 0xfffff9978f3e86800, len:
    98, name: "eth0")
17      7.662 ping/2736196 net:net_dev_start_xmit(name: "eth0", queue_mapping:
    45, skbaddr: 0xfffff9978f3e86800, protocol: 2048, len: 98, network_offset: 14,
    transport_offset_valid: 1, transport_offset: 34)
18      7.668 ping/2736196 net:net_dev_xmit(skbaddr: 0xfffff9978f3e86800, len: 98,
    name: "eth0")
19      27.396 ping/2736196 net:net_dev_queue(skbaddr: 0xfffff9978f3e87200, len:
    97, name: "lo")
20      27.402 ping/2736196 net:net_dev_start_xmit(name: "lo", skbaddr:
    0xfffff9978f3e87200, protocol: 2048, ip_summed: 3, len: 97, network_offset: 14,
    transport_offset_valid: 1, transport_offset: 34)
21      27.406 ping/2736196 net:netif_rx_entry(name: "lo", napi_id: 62, skbaddr:
    0xfffff9978f3e87200, protocol: 2048, ip_summed: 3, hash: 945309702, l4_hash: 1,
    len: 83, truesize: 768, mac_header_valid: 1, mac_header: 4294967282)
22      27.409 ping/2736196 net:netif_rx(skbaddr: 0xfffff9978f3e87200, len: 83,
    name: "lo")
23      27.412 ping/2736196 net:netif_rx_exit(skbaddr: 0xfffff9978f3e87200, len:
    83, name: "lo")
24      27.415 ping/2736196 net:net_dev_xmit(skbaddr: 0xfffff9978f3e87200, len: 97,
    name: "lo")
```

```

25      27.418 ping/2736196 net:netif_receive_skb(skbaddr: 0xfffff9978f3e87200,
26      len: 83, name: "lo")
27
28      --- baidu.com ping statistics ---
29      1 packets transmitted, 1 received, 0% packet loss, time 0ms
30      rtt min/avg/max/mdev = 19.672/19.672/19.672/0.000 ms

```

流控

影响发送的速度的几个buffer和queue，接收基本一样

Flow control is executed in several stages in the stack. **Figure 10** shows buffers used to transmit data. First, an application creates data and adds it to the send socket buffer. If there is no free space in the buffer, the system call is failed or the blocking occurs in the application thread. Therefore, the application data rate flowing into the kernel must be controlled by using the socket buffer size limit.

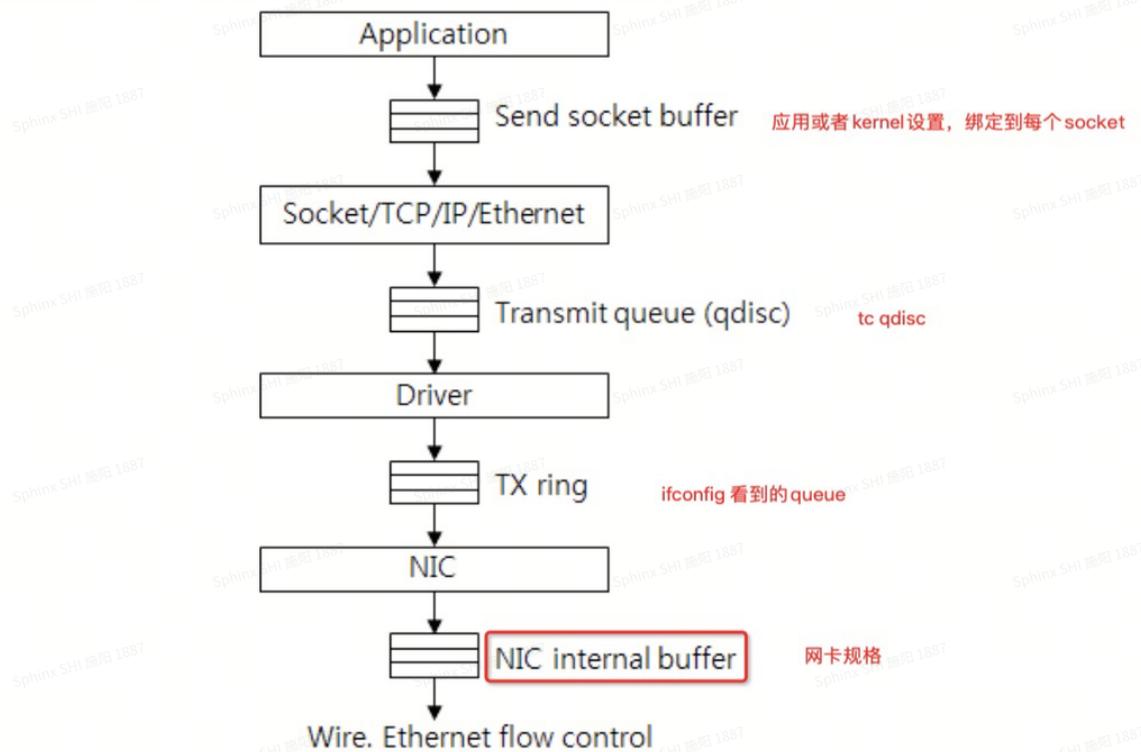
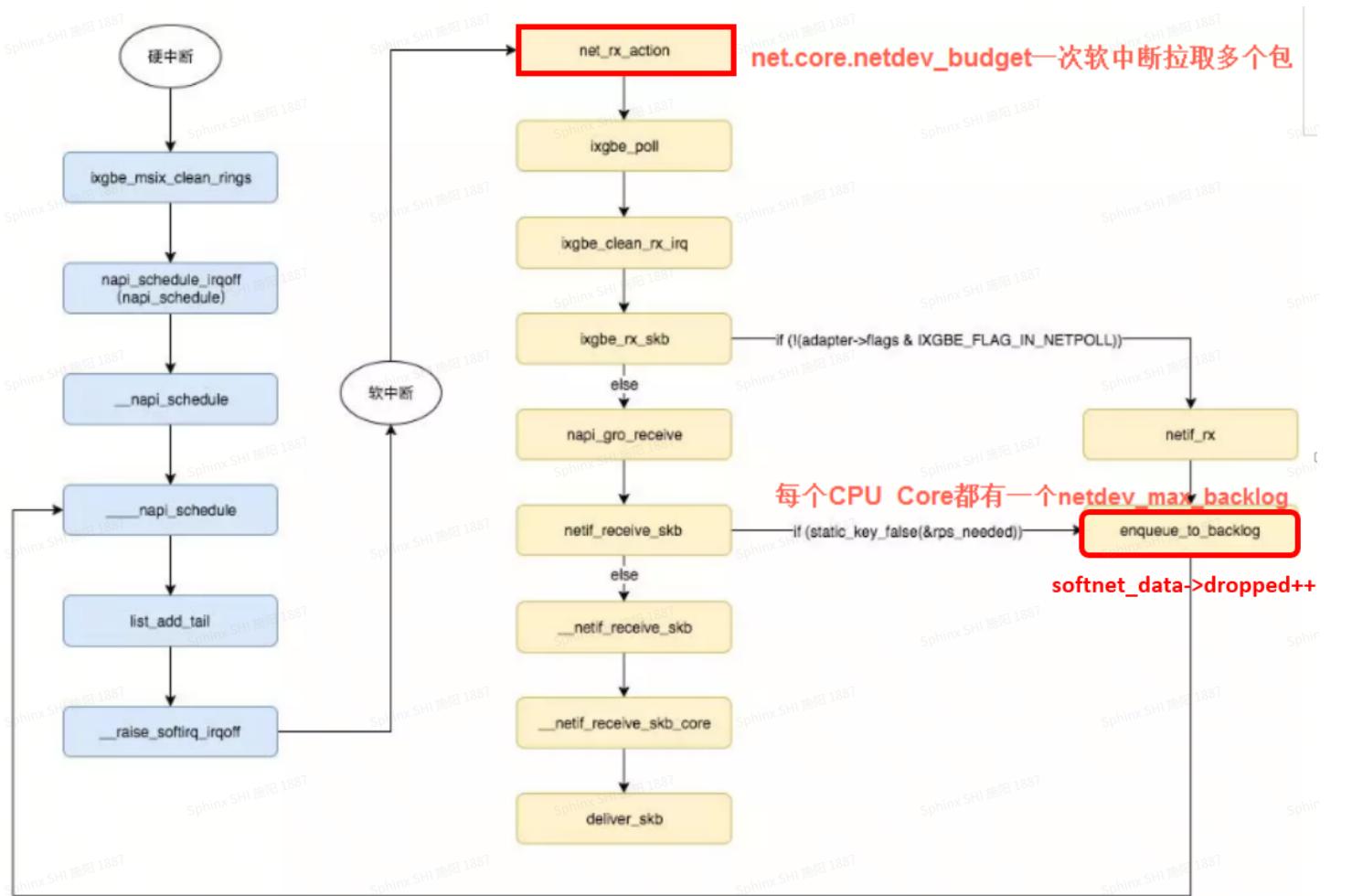


Figure 10: Buffers Related to Packet Transmission.

The TCP creates and sends packets to the driver through the transmit queue (qdisc). It is a typical FIFO queue type and the maximum length of the queue is the value of txqueuelen which can be checked by executing the ifconfig command. Generally, it is thousands of packets.

数据包传递流程图及参数

软中断NET_TX_SOFTIRQ的处理函数为net_tx_action，NET_RX_SOFTIRQ的为net_rx_action



在网络子系统初始化中为NET_RX_SOFTIRQ注册了处理函数net_rx_action。所以 `net_rx_action` 函数就会被执行到了。

ksoftirqd内核线程处理软中断

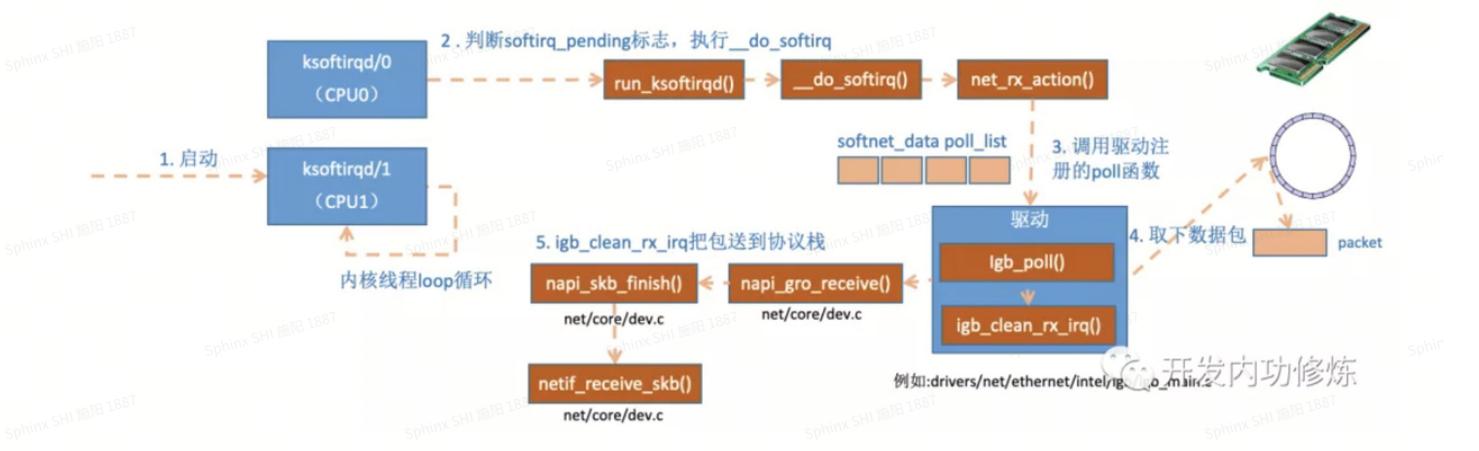
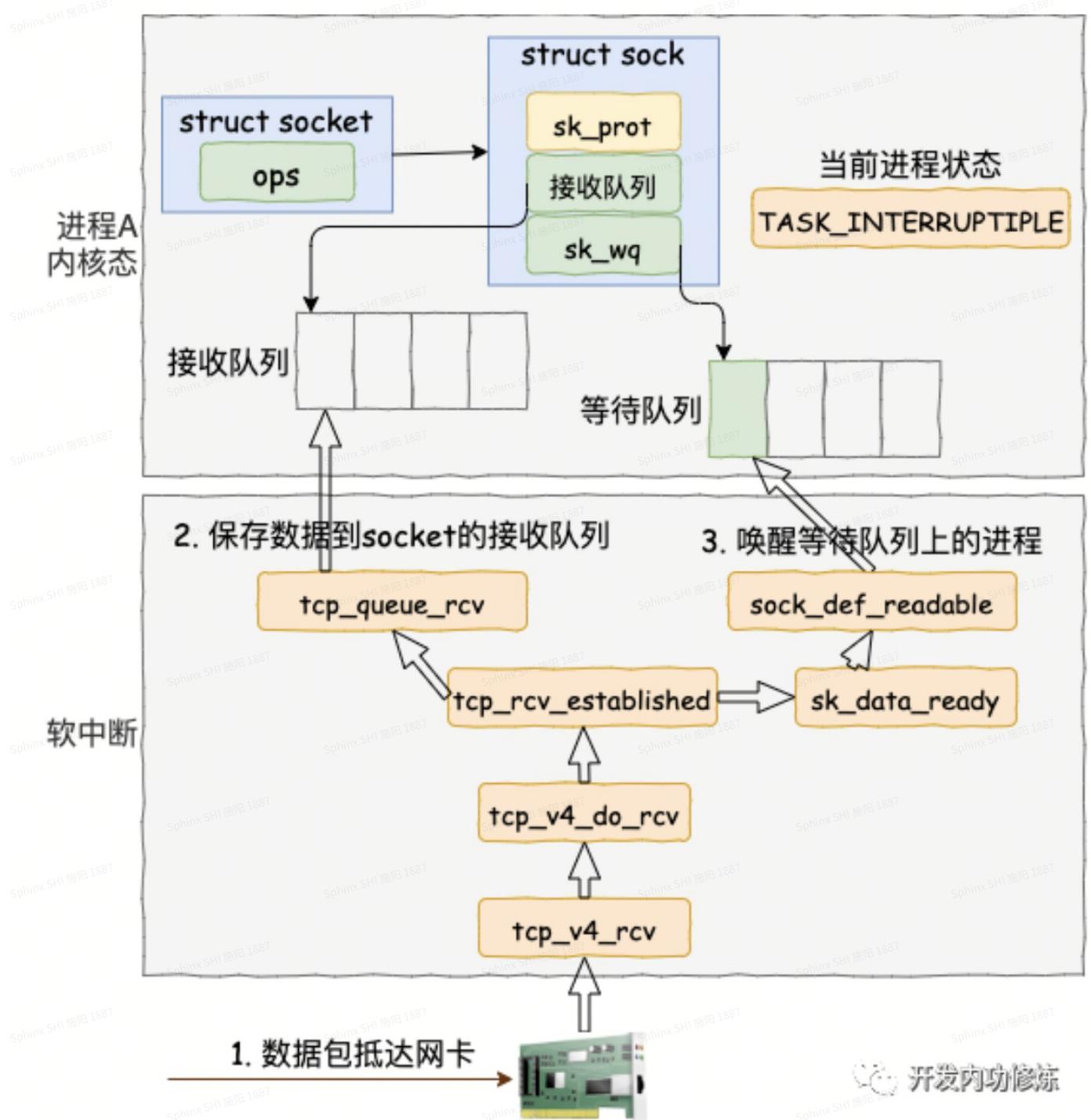


图9 ksoftirqd内核线程

这里需要注意一个细节，硬中断中设置软中断标记，和ksoftirq的判断是否有软中断到达，都是基于 `smp_processor_id()` 的。这意味着只要硬中断在哪个CPU上被响应，那么软中断也是在这个CPU上处理的。所以说，如果你发现你的Linux软中断CPU消耗都集中在一个核上的话，做法是要把调整硬中断的CPU亲和性，来将硬中断打散到不同的CPU核上去。

软中断（也就是 Linux 里的 ksoftirqd 进程）里收到数据包以后，发现是 tcp 的包的话就会执行到 `tcp_v4_rcv` 函数。如果是 ESTABLISHED 状态下的数据包，则最终会把数据拆出来放到对应 socket 的接收队列中。然后调用 `sk_data_ready` 来唤醒用户进程。



发送流程

1. 应用调 `sendmsg`
2. 数据拷贝到 `sk_write_queue` 上的最后一个 `skb` 中，如果该 `skb` 指向的数据区已经满了，则调用 `sk_stream_alloc_skb` 创建一个新的 `skb`，并挂到这个 `sk_write_queue` 上
3. TCP 分片 `skb_buff`
4. 根据 `tcp_wmem` 缓存需要发送的包



5. 构造TCP包头(src/dst port)
6. ipv4 调用 `tcp_write_xmit` 和 `tcp_transmit_skb`
7. `ip_queue_xmit`, 构建 ip 包头(获取目标ip和port, 找路由)
8. 进入 netfilter 流程 `nf_hook()`, `iptables`规则在这里生效
9. 路由流程 `POST_ROUTING`, `iptables` 的nat和mangle表会在这里设置规则, 对数据包进行一些修改
10. `ip_output` 分片
11. 进入L2 `dev_queue_xmit`, tc 网络流控在这里
12. 填入 txqueuelen 队列
13. 进入发送 Ring Buffer tx
14. 驱动触发软中断 soft IRQ (`NET_TX_SOFTIRQ`)

在传输层的出口函数 `tcp_transmit_skb` 中, 会对这个skb进行克隆¹ (`skb_clone`) , 克隆得到的子skb 和原先的父skb 指向共同的数据区。并且会把 `struct skb_shared_info` 的 `dataref` 的计数加一。

传输层以下各层处理的skb 实际就是这个克隆出来的skb, 而原先的skb保留在TCP连接的发送队列上。

克隆skb再经过协议栈层层处理后进入到驱动程序的RingBuffer 中。随后网卡驱动真正将数据发送出去, 当发送完成时, 由硬中断通知 CPU, 然后由中断处理程序来清理 RingBuffer中指向的skb。注意, 这里只释放了这个skb结构本身, 而skb指向的数据区, 由于dataref而不会被释放。要等到**TCP层接收到ACK后, 再释放父skb的同时, 释放数据区。**

比如 `ip_queue_xmit`发现无法路由到目标地址, 就会丢弃发送包, 这里丢弃的是克隆包, 原始包还在发送队列里, 所以TCP层就会在**定时器到期后进行重传**。

##

发包卡顿

内核从3.16开始有这样一个机制, 在生成的一个新的重传报文前, 先判断之前的报文的是否还在 `qdisc`里面, 如果在, 就避免生成一个新的报文。

也就是对内核而言这个包发送了但是没收到ack, 但实际这个包还在本机 `qdisc queue`或者 `driver queue`里, 所以没必要重传

对应的监控计数:

代码块

```
1 #netstat -s |grep -i spur
2 TCPSpuriousRtxHostQueues: 4163
```

这个发包过程在发送端实际抓不到这个包，因为还没有真正发送，而是在发送端的queue里排队，但是对上层应用来说包发完了（回包ack也不需要应用来感知），所以抓包看起来正常，就是应用感觉卡了（卡的原因还是包在发送端内核 queue 排队，一般是 pfifo_fast bug 和 bug2）

TCPSpuriousRtxHostQueues

Host queues (Qdisc + NIC) can hold packets so long that TCP can eventually retransmit a packet before the first transmit even left the host.

Its not clear right now if we could avoid this in the first place :

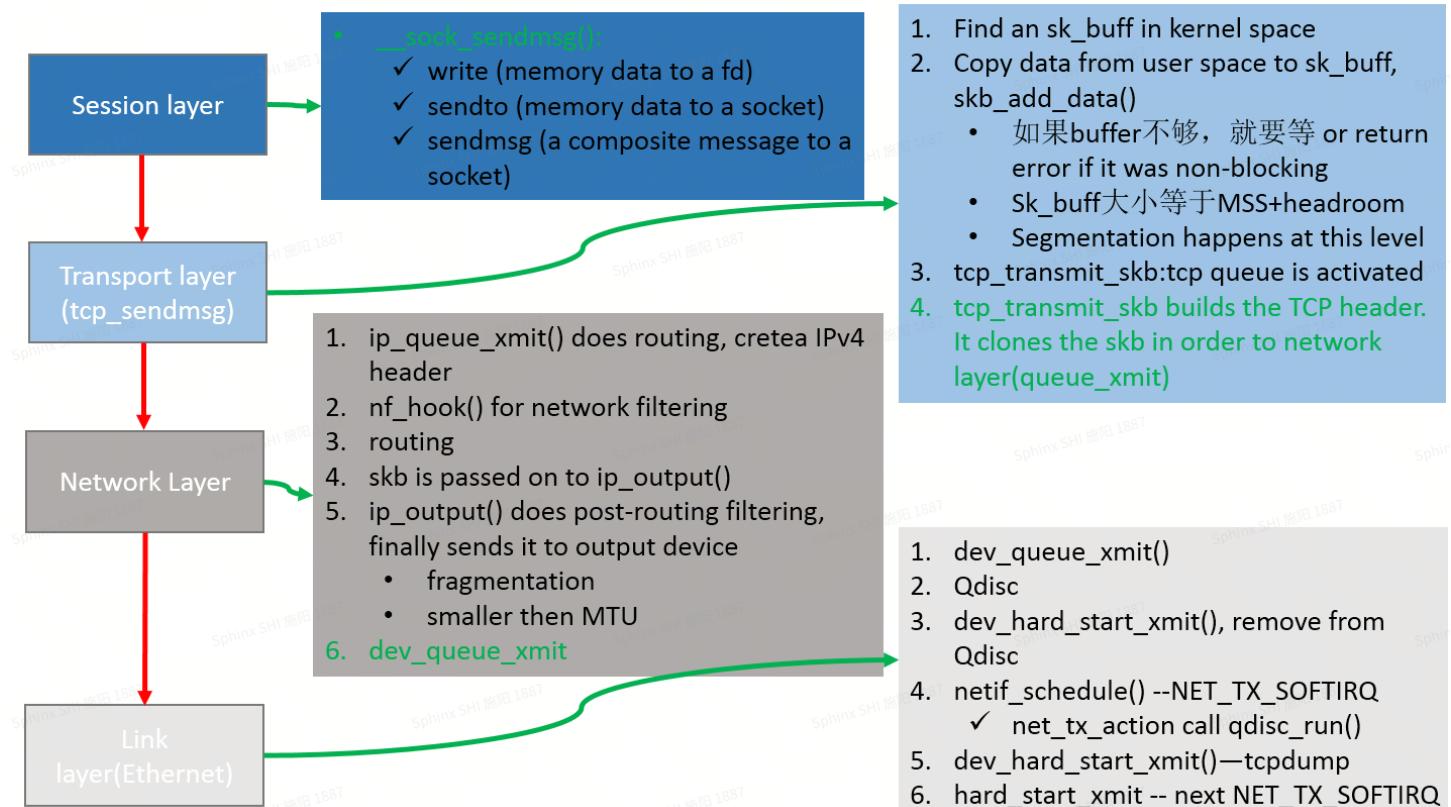
- We could arm RTO timer not at the time we enqueue packets, but at the time we TX complete them (`tcp_wfree()`)
- Cancel the sending of the new copy of the packet if prior one is still in queue.

This patch adds instrumentation so that we can at least see how often this problem happens. TCPSpuriousRtxHostQueues SNMP counter is incremented every time we detect the fast clone is not yet freed in `tcp_transmit_skb()`.

发包卡死

一个Linux 内核 bug 导致的 TCP连接卡死

TCP/IP 视角



发包流程对应源代码：

应用层

```
int main(){
    // 给用户返回结果
    send(fd, buf, sizeof(buf), 0);
}
```

系统调用

```
//file: net/socket.c
SYSCALL_DEFINE6(sendto, int, fd, ...
{
    //构造 msghdr 并赋值
    struct msghdr msg;
    * * *
    //发送数据
    sock_sendmsg(sock, &msg, len);
}
```

```
//file: net/socket.c
static inline int __sock_sendmsg_nosec(....)
{
    return sock->ops->sendmsg(iocb, sock, msg, size);
}
```

协议栈

```
//file: net/ipv4/af_inet.c
int inet_sendmsg(....)
{
    return sk->sk_prot->sendmsg(iocb, sk, msg, size);
}
```

```
//file: net/ipv4/tcp.c
int tcp_sendmsg(....)
{
    * * *
}
```

```
//file: net/ipv4/tcp_output.c
static int tcp_transmit_skb(....)
{
    //封装TCP头
    th = tcp_hdr(skb);
    th->source      = inet->inet_sport;
    th->dest        = inet->inet_dport;
    //调用网络层发送接口
    err = ip_queue_xmit(skb);
}
```

传输层

网络层

```
//file: net/ipv4/ip_output.c
int ip_queue_xmit(struct sk_buff *skb, struct flowi *fl)
{
    res = ip_local_out(skb);
}

//file: net/ipv4/ip_output.c
static inline int ip_finish_output2(struct sk_buff *skb)
{
    //继续向下游传递
    int res = dst_neigh_output(dst, neigh, skb);
}
```

链路层

```
//file: include/net/dst.h
static inline int dst_neigh_output(...)

*****
    return neigh_hh_output(hh, skb);
}

//file: include/net/neighbour.h
static inline int neigh_hh_output(...)
{
    ****
    skb_push(skb, hh_len);
    return dev_queue_xmit(skb);
}
```

网络设备子系统

```
//file: net/core/dev.c
int dev_queue_xmit(struct sk_buff *skb)
{
    //选择发送队列并获取 qdisc
    txq = netdev_pick_tx(dev, skb);
    q = rcu_dereference_bh(txq->qdisc);

    //则调用__dev_xmit_skb 继续发送
    rc = __dev_xmit_skb(skb, q, dev, txq);
}

//file: net/core/dev.c
int dev_hard_start_xmit(...)
{
    //获取设备的回调函数集合 ops
    const struct net_device_ops *ops = dev->netdev_ops;

    //调用驱动里的发送回调函数 ndo_start_xmit 将数据包传给网卡设备
    skb_len = skb->len;
```

```
rc = ops->ndo_start_xmit(skb, dev);
```

驱动程序

```
//file: drivers/net/ethernet/intel/igb/igb_main.c
static netdev_tx_t igb_xmit_frame(...)

{
    return igb_xmit_frame_ring(skb, ...);
}

netdev_tx_t igb_xmit_frame_ring(...)
{
    //获取TX Queue 中下一个可用缓冲区信息
    first = &tx_ring->tx_buffer_info[tx_ring->next_to_use];
    first->skb = skb;
    first->bytecount = skb->len;

    //igb_tx_map 函数准备给设备发送的数据。
    igb_tx_map(tx_ring, first, hdr_len);
}
```

硬件



`net.core.dev_weight` 用来调整 `__qdisc_run` 的循环处理权重，调大后也就是 `__netif_schedule` 更多的被调用；

发包默认是系统调用完成的(占用 sy cpu)，只有在包太多，为了避免系统调用长时间占用 CPU 导致应用层卡顿，这个时候内核给了发包时间一个quota(`net.core.dev_weight` 参数来控制)，用完后即使包没发送完也退出发包的系统调用，队列中未发送完的包留待 tx-softirq 来发送(这是占用 si cpu)；

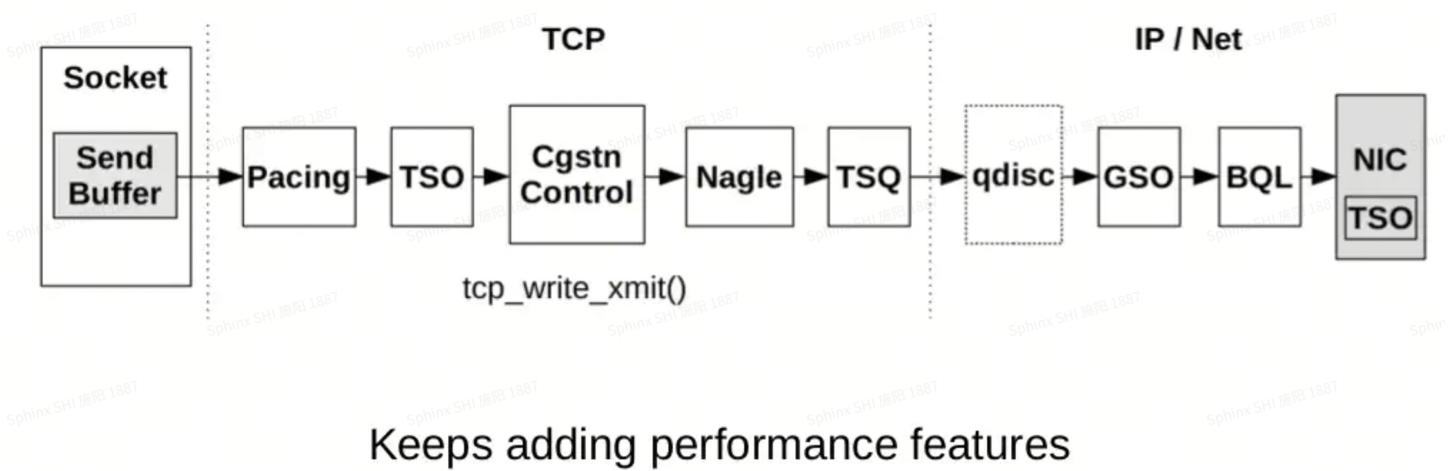
tcp在做`tcp_sendmsg`的时候会将应用层msg做copy到内核层的skb，在调用网络层执行`tcp_transmit_skb`会将这个skb再次copy交给网络层，内核态的skb继续保留直到收到ack。

`tcp_transmit_skb`还会设置tcp头，在skb中tcp头、ip头内存都预留好了，只需要填写内容。

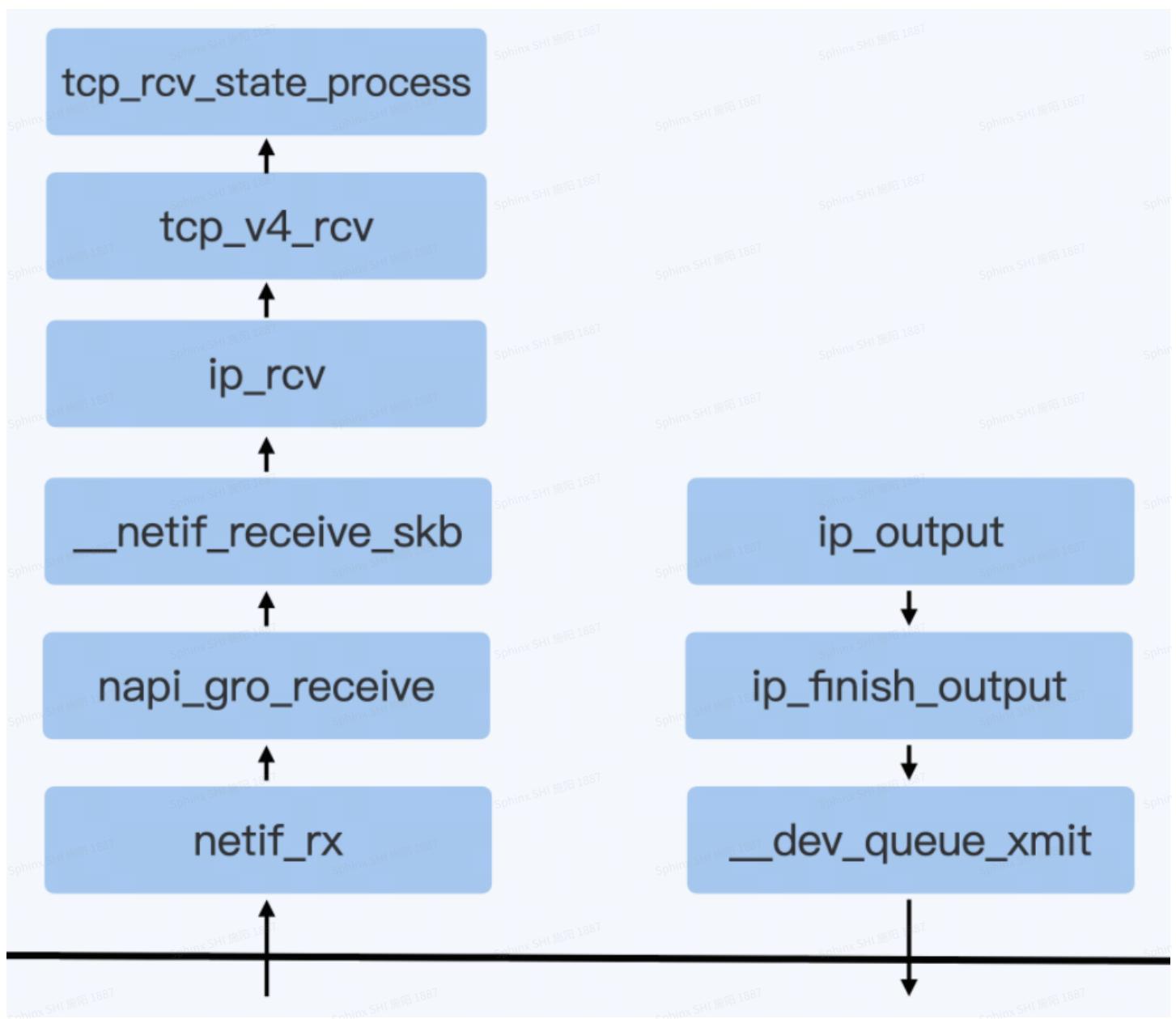
然后就是ip层，主要是分包、路由控制，然后就是netfilter(比如iptables规则匹配)。

再然后进入neighbour(arp)，获取mac后进入网络层用 `sudo ifconfig eth0 txqueuelen **` 来控制qdisc 发送队列长度

Linux TCP send path



粗略汇总一下进出堆栈：



Data Transmission

As indicated by its name, a network stack has many layers. The following **Figure 1** shows the layer types.

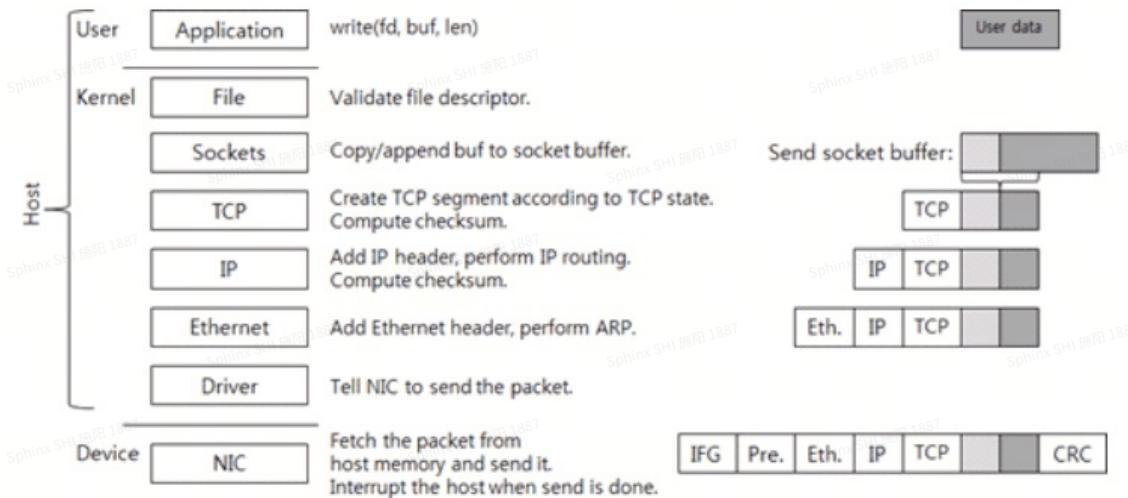


Figure 1: Operation Process by Each Layer of TCP/IP Network Stack for Data Transmission.

软中断

一般net_rx 远大于net_tx, 如下所示，这是因为每个包发送完成后还需要清理回收内存(释放 skb)，这是通过硬中断触发 rx-softirq 来完成的，无论是收包、还是发送包完毕都是触发这个rx-softirq。

代码块							
1	cut /proc/softirqs -c 1-255						
2	CPU0	CPU1	CPU2	CPU3	CPU4	CPU5	
	CPU6	CPU7	CPU8	CPU9	CPU10	CPU11	CPU12
	CPU13	CPU14	CPU15	CPU16	CPU17	CPU18	CPU19
	CPU20	CPU2					
3	HI:	0	0	0	0	0	0
	0	0	0	0	0	0	0
	0	0	0	0	0	0	0
4	TIMER:	1308419	2694651	2954548	680581	3468824	609308
	2398055	551832	3076612	543148	2042747	376586	1811968
	623665	1997474	719990	2586022	3948569	3569294	632179
	2668121	472573					
5	NET_TX:	10	0	7	0	4	7
	2	2	3	0	2	1	1
	6	2	7	10	6	5	0
	0	2					
6	NET_RX:	136752	37368	133183	22494	344006	232432
	249291	162459	267769	231343	317403	313842	298159
	366732	258942	141110	1123730	1069771	910948	1043782
	795582	819413					

7	BLOCK:	204	4	48370	1	37	16
		37	46	24	16	8	25
		0	10	327	7327	39755	21340
		12965	1323				2180
8	IRQ_POLL:	0	0	0	0	0	0
		0	0	0	0	0	0
		0	0	0	0	0	0
		0	0				
9	TASKLET:	762	144	770	74	798	142
		1389	117	787	95	802	114
		58	641	212	10306	13790	17119
		6768	17813				16027
10	SCHED:	21466955	9826444	20743198	4547285	22001350	4403487
		14816005	7406282	18036122	6192058	14010769	4768976
		7078775	15795592	5821156	24183911	9859222	18517264
		24034567	5564654				6448838
11	HRTIMER:	0	1	0	0	10	1
		3712	1	7	5	5	28
		108	0	9	61	9	141
		19	63				287
12	RCU:	16427859	8076269	16388325	4160693	17336115	4052909
		12267892	6447703	14637266	5807650	11646213	4522785
		6833462	13340509	5589879	19115956	9163734	14499348
		18697262	5233588				6063949

发送的时候如果 net.core.dev_weight 配额够的话直接通过系统调用就将包发送完毕，不需要触发软中断

内核相关参数

Ring Buffer

Ring Buffer位于NIC和IP层之间，是一个典型的FIFO（先进先出）环形队列。Ring Buffer没有包含数据本身，而是包含了指向sk_buff (socket kernel buffers) 的描述符。

可以使用ethtool -g eth0查看当前Ring Buffer的设置：

代码块

```

1 $sudo ethtool -g eth0
2 Ring parameters for eth0:
3 Pre-set maximums:
4 RX:          256
5 RX Mini:     0
6 RX Jumbo:    0
7 TX:          256

```

```
8 Current hardware settings:  
9 RX: 256  
10 RX Mini: 0  
11 RX Jumbo: 0  
12 TX: 256
```

上面的例子是一个小规格的ECS，接收队列、传输队列都为256。

代码块

```
1 $sudo ethtool -g eth0  
2 Ring parameters for eth0:  
3 Pre-set maximums:  
4 RX: 4096  
5 RX Mini: 0  
6 RX Jumbo: 0  
7 TX: 4096  
8 Current hardware settings:  
9 RX: 4096  
10 RX Mini: 0  
11 RX Jumbo: 0  
12 TX: 512
```

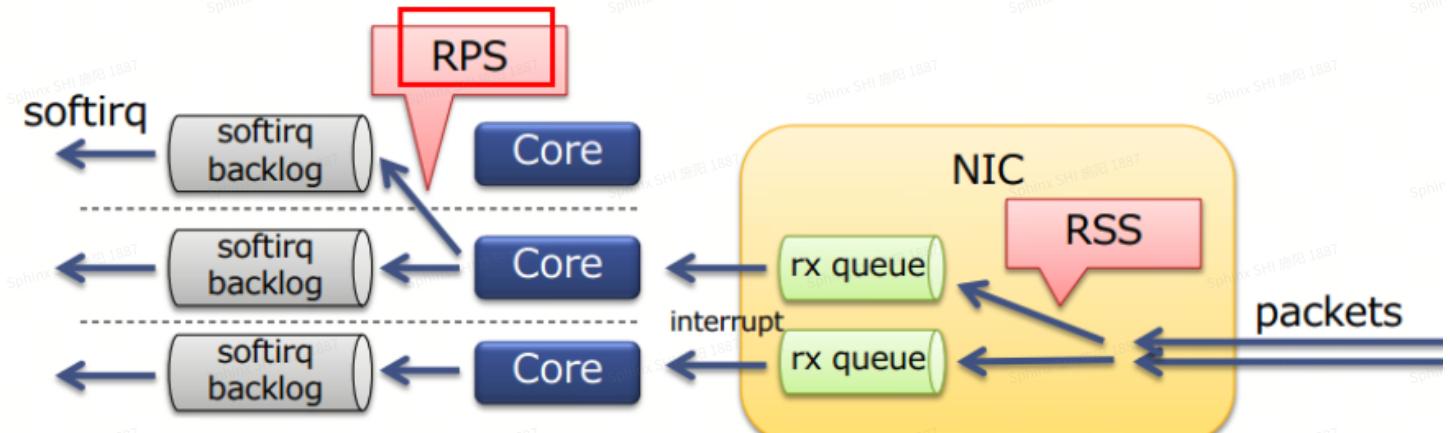
这是一台物理机，接收队列为4096，传输队列为512。接收队列已经调到了最大，传输队列还可以调大。**队列越大丢包的可能越小，但数据延迟会增加**

调整 Ring Buffer 队列数量

代码块

```
1 ethtool -l eth0  
2 Channel parameters for eth0:  
3 Pre-set maximums:  
4 RX: 0  
5 TX: 0  
6 Other: 1  
7 Combined: 8  
8 Current hardware settings:  
9 RX: 0  
10 TX: 0  
11 Other: 1  
12 Combined: 8  
13  
14 sudo ethtool -L eth0 combined 8  
15 sudo ethtool -L eth0 rx 8
```

网卡多队列就是指的有多个RingBuffer，每个RingBuffer可以由一个core来处理



网卡各种统计数据查看

代码块

```
1 ethtool -S eth0 | grep errors
2 ethtool -S eth0 | grep rx_ | grep errors //查看网卡是否丢包，一般是ring buffer太小
3 //监控
4 ethtool -S eth0 | grep -e "err" -e "drop" -e "over" -e "miss" -e "timeout" -e
  "reset" -e "restart" -e "collis" -e "over" | grep -v "\: 0"
```

网卡进出队列大小调整

代码块

```
1 //查看目前的进出队列大小
2 ethtool -g eth0
3 //修改进出队列
4 ethtool -G eth0 rx 8192 tx 8192
```

要注意如果设置的值超过了允许的最大值，用默认的最大值，一些ECS之类的虚拟机、容器就不允许修改这个值。

txqueuelen

ifconfig 看到的 txqueuelen 跟Ring Buffer是两个东西，IP协议下面就是 txqueuelen，txqueuelen下面才到Ring Buffer.

常用的tc qdisc、netfilter就是在txqueuelen这一环节。qdisc 的队列长度是我们用 ifconfig 来看到的 txqueuelen

发送队列就是指的这个txqueuelen，和网卡关联着。

而每个Core接收队列由内核参数： net.core.netdev_max_backlog来设置

代码块

```
1 //当前值通过ifconfig可以查看到，修改：  
2 ifconfig eth0 txqueuelen 2000  
3 //监控  
4 ip -s link  
5 ip -s link show enp2s0f0
```

如果txqueuelen 太小导致数据包被丢弃的情况，这类问题可以通过下面这个命令来观察：

代码块

```
1 sudo ip -s -s link ls dev eth0  
2 2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP mode  
    DEFAULT group default qlen 1000  
    link/ether 52:54:00:af:55:78 brd ff:ff:ff:ff:ff:ff  
    RX: bytes packets errors dropped missed mcast  
    5 118647165658 93832250 0 0 0 0  
    RX errors: length crc frame fifo overrun  
    7 0 0 0 0 0  
    TX: bytes packets errors dropped carrier collsns  
    9 17319282178 18958102 0 0 0 0  
    TX errors: aborted fifo window heartbt transns  
    11 0 0 0 0 2  
    altname enp0s5  
    altname ens5
```

如果观察到 dropped 这一项不为 0，那就有可能是 txqueuelen 太小导致的。当遇到这种情况时，你就需要增大该值了，比如增加 eth0 这个网络接口的 txqueuelen, `ifconfig eth0 txqueuelen 2000`

Interrupt Coalescence (IC) - rx-usecs, tx-usecs, rx-frames, tx-frames (hardware IRQ)

可以通过降低终端的频率，也就是合并硬中断来提升处理网络包的能力，当然这是以增大网络包的延迟为代价。

代码块

```
1 ethtool -c eth0  
2 Coalesce parameters for eth0:  
3 Adaptive RX: n/a TX: n/a  
4 stats-block-usecs: n/a  
5 sample-interval: n/a  
6 pkt-rate-low: n/a
```

```
7  pkt-rate-high: n/a
8
9  rx-usecs: n/a
10 rx-frames: 1
11 rx-usecs-irq: n/a
12 rx-frames-irq: n/a
13
14 tx-usecs: n/a
15 tx-frames: 1
16 tx-usecs-irq: n/a
17 tx-frames-irq: n/a
18
19 rx-usecs-low: n/a
20 rx-frame-low: n/a
21 tx-usecs-low: n/a
22 tx-frame-low: n/a
23
24 rx-usecs-high: n/a
25 rx-frame-high: n/a
26 tx-usecs-high: n/a
27 tx-frame-high: n/a
28
29 CQE mode RX: n/a TX: n/a
30
```

- Adaptive RX: 自适应中断合并，网卡驱动自己判断啥时候该合并啥时候不合并
- rx-usecs: 当过这么长时间过后，一个RX interrupt就会被产生。How many usecs to delay an RX interrupt after a packet arrives.
- rx-frames: 当累计接收到这么多个帧后，一个RX interrupt就会被产生。Maximum number of data frames to receive before an RX interrupt.
- rx-usecs-irq : How many usecs to delay an RX interrupt while an interrupt is being serviced by the host.
- rx-frames-irq : Maximum number of data frames to receive before an RX interrupt is generated while the system is servicing an interrupt.

Ethtool 绑定端口

ntuple filtering for steering network flows

一些网卡支持 “ntuple filtering” 特性。该特性允许用户（通过 ethtool）指定一些参数来在硬件上过滤收到的包，然后将其直接放到特定的 RX queue。例如，用户可以指定到特定目端口的 TCP 包放到 RX queue 1。

Intel 的网卡上这个特性叫 Intel Ethernet Flow Director，其他厂商可能也有他们的名字，这些都是出于市场宣传原因，底层原理是类似的。

我们后面会看到，ntuple filtering 是一个叫 Accelerated Receive Flow Steering(aRFS) 功能的核心部分之一，后者使得 ntuple filtering 的使用更加方便。

这个特性适用的场景：最大化数据本地性 (data locality)，以增加 CPU 处理网络数据时的缓存命中率。例如，考虑运行在 80 口的 web 服务器：

1. webserver 进程运行在 80 口，并绑定到 CPU 2
2. 和某个 RX queue 关联的硬中断绑定到 CPU 2
3. 目的端口是 80 的 TCP 流量通过 ntuple filtering 绑定到 CPU 2
4. 接下来所有到 80 口的流量，从数据包进来到数据到达用户程序的整个过程，都由 CPU 2 处理
5. 仔细监控系统的缓存命中率、网络栈的延迟等信息，以验证以上配置是否生效

检查 ntuple filtering 特性是否打开：

代码块

```
1 sudo ethtool -k eth0
2 Features for eth0:
3   rx-checksumming: on [fixed]
4     tx-checksumming: on
5       tx-checksum-ipv4: off [fixed]
6       tx-checksum-ip-generic: on
7       tx-checksum-ipv6: off [fixed]
8       tx-checksum-fcoe-crc: off [fixed]
9       tx-checksum-sctp: off [fixed]
10    scatter-gather: on
11      tx-scatter-gather: on
12      tx-scatter-gather-fraglist: off [fixed]
13    tcp-segmentation-offload: on
14      tx-tcp-segmentation: on
15      tx-tcp-ecn-segmentation: off [fixed]
16      tx-tcp-mangleid-segmentation: off
17      tx-tcp6-segmentation: off [fixed]
18    generic-segmentation-offload: on
19    generic-receive-offload: on
20  large-receive-offload: off [fixed]
21  rx-vlan-offload: off [fixed]
22  tx-vlan-offload: off [fixed]
23  ntuple-filters: off [fixed]
24  receive-hashing: off [fixed]
25  highdma: on [fixed]
26  rx-vlan-filter: on [fixed]
27  vlan-challenged: off [fixed]
```

```
28 tx-lockless: off [fixed]
29 netns-local: off [fixed]
30 tx-gso-robust: on [fixed]
31 tx-fcoe-segmentation: off [fixed]
32 tx-gre-segmentation: off [fixed]
33 tx-gre-csum-segmentation: off [fixed]
34 tx-ipxip4-segmentation: off [fixed]
35 tx-ipxip6-segmentation: off [fixed]
36 tx-udp_tnl-segmentation: off [fixed]
37 tx-udp_tnl-csum-segmentation: off [fixed]
38 tx-gso-partial: off [fixed]
39 tx-tunnel-remcsum-segmentation: off [fixed]
40 tx-sctp-segmentation: off [fixed]
41 tx-esp-segmentation: off [fixed]
42 tx-udp-segmentation: off [fixed]
43 tx-gso-list: off [fixed]
44 fcoe-mtu: off [fixed]
45 tx-nocache-copy: off
46 loopback: off [fixed]
47 rx-fcs: off [fixed]
48 rx-all: off [fixed]
49 tx-vlan-stag-hw-insert: off [fixed]
50 rx-vlan-stag-hw-parse: off [fixed]
51 rx-vlan-stag-filter: off [fixed]
52 l2-fwd-offload: off [fixed]
53 hw-tc-offload: off [fixed]
54 esp-hw-offload: off [fixed]
55 esp-tx-csum-hw-offload: off [fixed]
56 rx-udp_tunnel-port-offload: off [fixed]
57 tls-hw-tx-offload: off [fixed]
58 tls-hw-rx-offload: off [fixed]
59 rx-gro-hw: off
60 tls-hw-record: off [fixed]
61 rx-gro-list: off
62 macsec-hw-offload: off [fixed]
63 rx-udp-gro-forwarding: off
64 hsr-tag-ins-offload: off [fixed]
65 hsr-tag-rm-offload: off [fixed]
66 hsr-fwd-offload: off [fixed]
67 hsr-dup-offload: off [fixed]
```

可以看到，上面的 ntuple 是关闭的。

打开：

代码块

```
1 蔡阳 1887 sudo ethtool -K eth0 ntuple on
```

打开 ntuple filtering 功能，并确认打开之后，可以用 `ethtool -u` 查看当前的 ntuple rules：

代码块

```
1 sudo ethtool -u eth0
2 40 RX rings available
3 Total 0 rules
```

可以看到当前没有 rules。

我们来加一条：目的端口是 80 的放到 RX queue 2：

代码块

```
1 $ sudo ethtool -U eth0 flow-type tcp4 dst-port 80 action 2
2
3 删除
4 ethtool -U eth0 delete 1023
```

也可以用 ntuple filtering 在硬件层面直接 drop 某些 flow 的包。当特定 IP 过来的流量太大时，这种功能可能会派上用场。更多关于 ntuple 的信息，参考 ethtool man page。

软中断合并 GRO

GRO和硬中断合并的思想很类似，不过阶段不同。硬中断合并是在中断发起之前，而GRO已经到了软中断上下文中了。

如果应用中是大文件的传输，大部分包都是一段数据，不用GRO的话，会每次都把一个小包传送到协议栈（IP接收函数、TCP接收）函数中进行处理。开启GRO的话，Linux就会智能进行包的合并，之后将一个大包传给协议处理函数。这样CPU的效率也是就提高了。

代码块

```
1 ethtool -k eth0 | grep generic-receive-offload
2 generic-receive-offload: on
```

如果你的网卡驱动没有打开GRO的话，可以通过如下方式打开。

代码块

```
1 ethtool -K eth0 gro on
```

这是收包，发包对应参数是GSO

ifconfig 监控指标

- RX overruns: overruns意味着数据包没到Ring Buffer就被网卡物理层给丢弃了，而CPU无法及时的处理中断是造成Ring Buffer满的原因之一，例如中断分配的不均匀。或者Ring Buffer太小导致的（很少见），overruns数量持续增加，建议增大Ring Buffer，使用ethtool - G 进行设置。
- RX dropped: 表示数据包已经进入了Ring Buffer，但是由于内存不够等系统原因，导致在拷贝到内存的过程中被丢弃。如下四种情况导致dropped: Softnet backlog full (pfmemalloc && !skb_pfmemalloc_protocol(skb)-分配内存失败)；Bad / Unintended VLAN tags；Unknown / Unregistered protocols；IPv6 frames
- RX errors: 表示总的收包的错误数量，这包括 too-long-frames 错误，Ring Buffer 溢出错误，crc 校验错误，帧同步错误，fifo overruns 以及 missed pkg 等等。

overruns

当驱动处理速度跟不上网卡收包速度时，驱动来不及分配缓冲区，NIC接收到的数据包无法及时写到sk_buffer，就会产生堆积，当NIC内部缓冲区写满后，就会丢弃部分数据，引起丢包。这部分丢包为rx_fifo_errors，在/proc/net/dev中体现为fifo字段增长，在ifconfig中体现为overruns指标增长。

监控指标 /proc/net/softnet_stat

Important details about `/proc/net/softnet_stat`:

- Each line of `/proc/net/softnet_stat` corresponds to a `struct softnet_data` structure, of which there is 1 per CPU.
- The values are separated by a single space and are displayed in hexadecimal
- The first value, `sd->processed`, is the number of network frames processed. This can be more than the total number of network frames received if you are using ethernet bonding. There are cases where the ethernet bonding driver will trigger network data to be re-processed, which would increment the `sd->processed` count more than once for the same packet.
- The second value, `sd->dropped`, is the number of network frames dropped because there was no room on the processing queue. More on this later.
- The third value, `sd->time_squeeze`, is (as we saw) the number of times the `net_rx_action` loop terminated because the budget was consumed or the time limit was reached, but more work could have been. Increasing the `budget` as explained earlier can help reduce this. **time_squeeze** 计数在内核中只有一个地方会更新（比如内核 5.10），如果看到监控中有 time_squeeze 升高，那一定就是执行到了以上 budget 用完的逻辑
- The next 5 values are always 0.

- The ninth value, `sd->cpu_collision`, is a count of the number of times a collision occurred when trying to obtain a device lock when transmitting packets. This article is about receive, so this statistic will not be seen below.
- The tenth value, `sd->received_rps`, is a count of the number of times this CPU has been woken up to process packets via an **Inter-processor Interrupt**
- The last value, `flow_limit_count`, is a count of the number of times the flow limit has been reached. Flow limiting is an optional **Receive Packet Steering** feature that will be examined shortly.

对应的代码实现：

代码块

```

1 // https://github.com/torvalds/linux/blob/v5.10/net/core/net-procfs.c#L172
2 static int softnet_seq_show(struct seq_file *seq, void *v) {
3     struct softnet_data *sd = v;
4
5     seq_printf(seq,
6                 "%08x %08x %08x\n",
7                 sd->processed, sd->dropped, sd->time_squeeze, 0,
8                 0, 0, 0, 0, /* was fastroute */
9                 0, /* was cpu_collision */
10                sd->received_rps, flow_limit_count,
11                softnet_backlog_len(sd), (int)seq->index);
12
13 }
```

net.core.netdev_budget

一次软中断(ksoftirqd进程)能处理包的上限，有就多处理，处理到300个了一定要停下来让CPU能继续其它工作。单次poll收包是所有注册到这个CPU的NAPI变量收包数量之和不能大于这个阈值。

代码块

```

1 sysctl net.core.netdev_budget //3.10 kernel默认300, The default value of the
budget is 300. This will cause the SoftIRQ process to drain 300 messages from
the NIC before getting off the CPU
```

如果 /proc/net/softnet_stat 第三列一直在增加的话需要，表示SoftIRQ 获取的CPU时间太短，来不及处理足够多的网络包，那么需要增大这个值，**当这个值太大的话有可能导致包到了内核但是应用(userspace) 抢不到时间片来读取这些packet。**

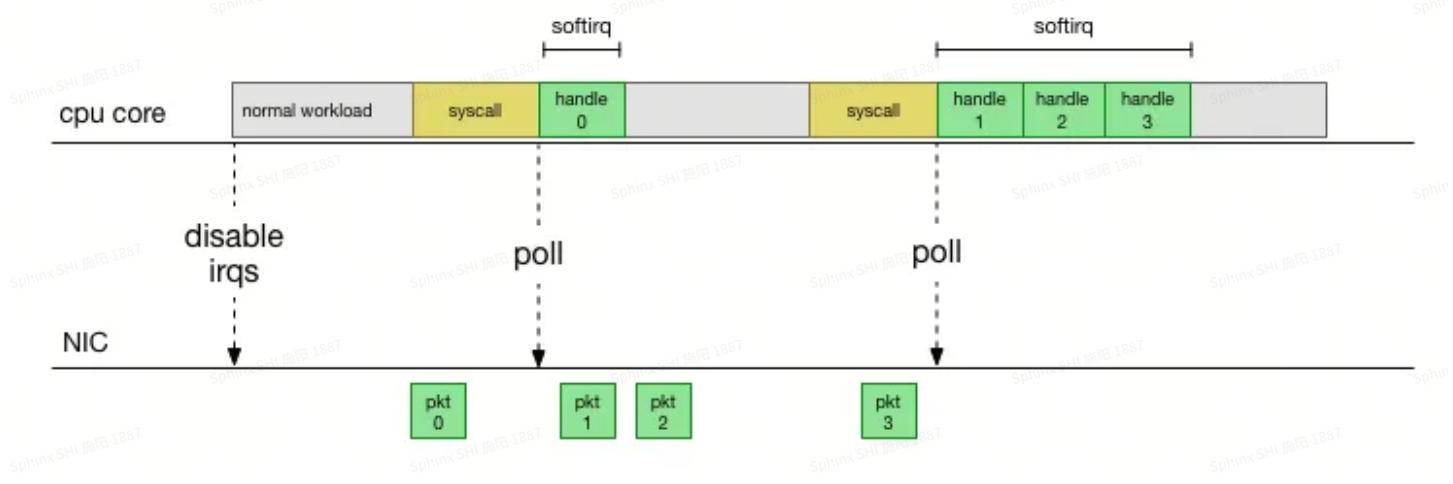
增大和查看 net.core.netdev_budget

```
sysctl -a | grep net.core.netdev_budget
```

```
sysctl -w net.core.netdev_budget=400 //临时性增大
```

早期的时候网卡一般是10Mb的，现在基本都是10Gb的了，还是每一次软中断、上下文切换只处理一个包的话代价太大，需要改进性能。于是引入的NAPI，一次软中断会poll很多packet

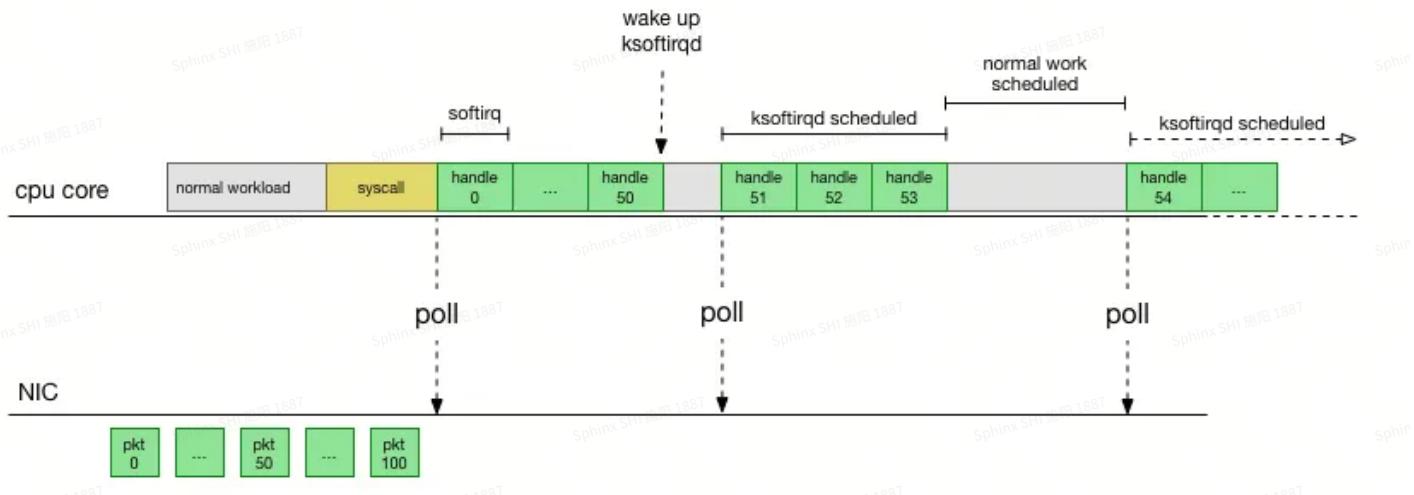
NAPI polling in softirq at end of syscalls



来源 This is much faster, but brings up another problem. What happens if we have so many packets to process that we spend all our time processing packets from the NIC, but we never have time to let the userspace processes actually drain those queues (read from TCP connections, etc.)? Eventually the queues would fill up, and we'd start dropping packets. To try and make this fair, the kernel limits the amount of packets processed in a given softirq context to a certain budget. Once this budget is exceeded, it wakes up a separate thread called `ksoftirqd` (you'll see one of these in `ps` for each core) which processes these softirqs outside of the normal syscall/interrupt path. This thread is scheduled using the standard process scheduler, which already tries to be fair.

于是在Poll很多packet的时候有可能网卡队列一直都有包，那么导致这个Poll动作无法结束，造成应用一直在卡住状态，于是可以通过`netdev_max_backlog`来设置Poll多少Packet后停止Poll以响应用户请求。

NAPI polling crossing threshold to schedule ksoftirqd



一旦出现slow syscall (如上图黄色部分慢) 就会导致packet处理被延迟。

发送包的时候系统调用循环发包，占用sy，只有当发包系统quota用完的时候，循环退出，剩下的包通过触发软中断的形式继续发送，此时占用si

netdev_max_backlog

The netdev_max_backlog is a queue within the Linux kernel where traffic is stored after reception from the NIC, but before processing by the protocol stacks (IP, TCP, etc). There is one backlog queue per CPU core.

如果 /proc/net/softnet_stat 第二列一直在增加的话表示netdev backlog queue overflows. 需要增大 netdev_max_backlog

增大和查看 netdev_max_backlog:

sysctl -a |grep netdev_max_backlog

sysctl -w net.core.netdev_max_backlog=1024 //临时性增大

netdev_max_backlog(接收)和txqueuelen(发送)相对应

softnet_stat

关于 /proc/net/softnet_stat 的重要细节:

- 每一行代表一个 `struct softnet_data` 变量。因为每个 CPU core 都有一个该变量，所以每行其实代表一个 CPU core
- 每列用空格隔开，数值用 16 进制表示
- 第一列 `sd->processed`，是处理的网络帧的数量。如果你使用了 ethernet bonding，那这个值会大于总的网络帧的数量，因为 ethernet bonding 驱动有时会触发网络数据被重新处理 (re-processed)
- 第二列，`sd->dropped`，是因为处理不过来而 drop 的网络帧数量。后面会展开这一话题

5. 第三列，`sd->time_squeeze`，前面介绍过了，由于 budget 或 time limit 用完而退出
`net_rx_action` 循环的次数

6. 接下来的 5 列全是 0

7. 第九列，`sd->cpu_collision`，是为了发送包而获取锁的时候有冲突的次数

8. 第十列，`sd->received_rps`，是这个 CPU 被其他 CPU 唤醒去收包的次数

9. 最后一列，`flow_limit_count`，是达到 flow limit 的次数。flow limit 是 RPS 的特性。

TCP协议栈Buffer

代码块

```
1 sysctl -a | grep net.ipv4.tcp_rmem // receive
2 sysctl -a | grep net.ipv4.tcp_wmem // send
3 //监控
4 cat /proc/net/sockstat
```

接收Buffer

代码块

```
1 $netstat -sn | egrep "prune|collap"; sleep 30; netstat -sn | egrep
  "prune|collap"
2 17671 packets pruned from receive queue because of socket buffer overrun
3 18671 packets pruned from receive queue because of socket buffer overrun
```

如果“pruning”一直在增加很有可能是程序中调用了 `setsockopt(SO_RCVBUF)` 导致内核关闭了动态调整功能，或者压力大，缓存不够了。具体Case：<https://blog.cloudflare.com/the-story-of-one-latency-spike/>

nstat也可以看到比较多的数据

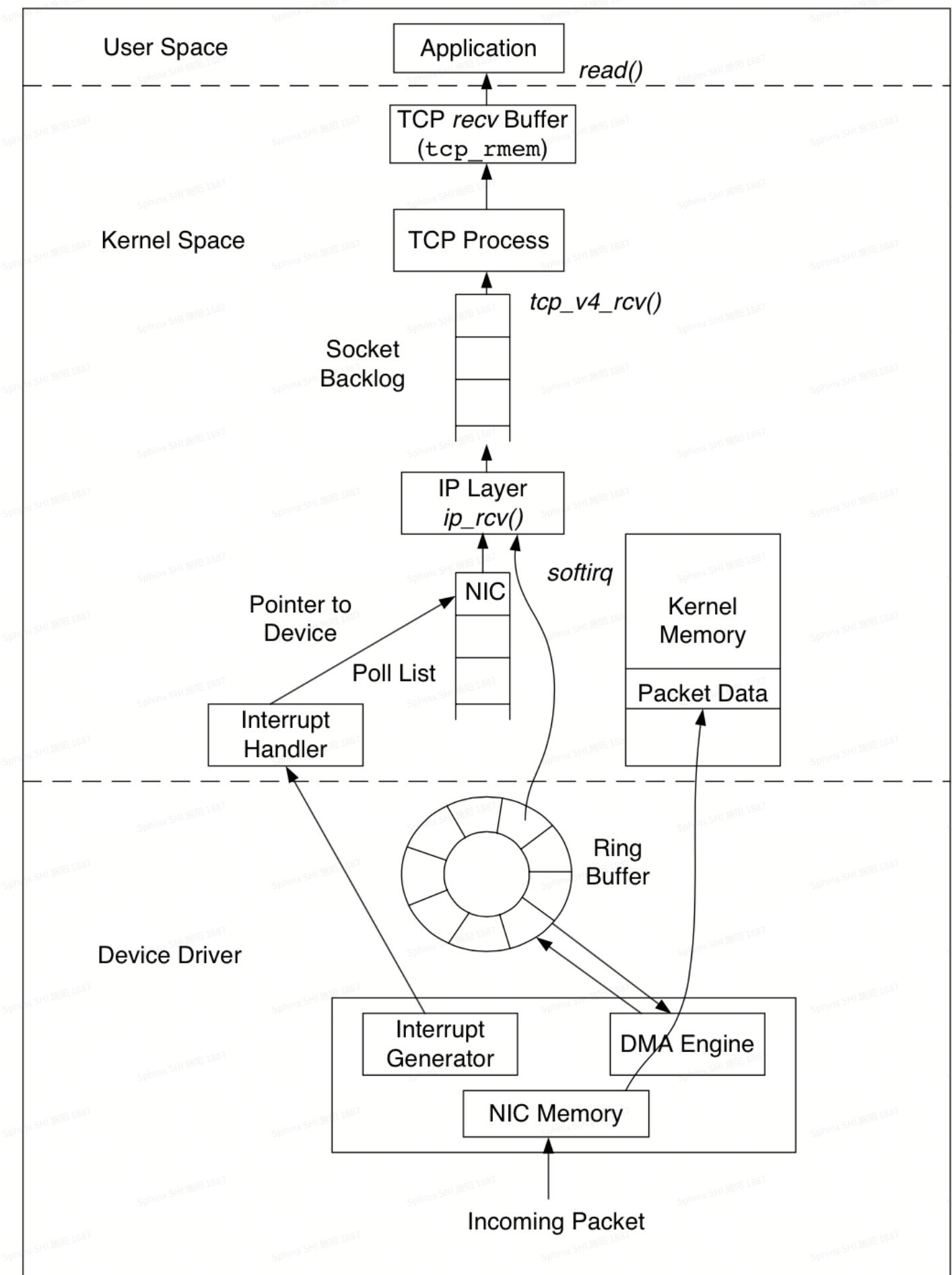
代码块

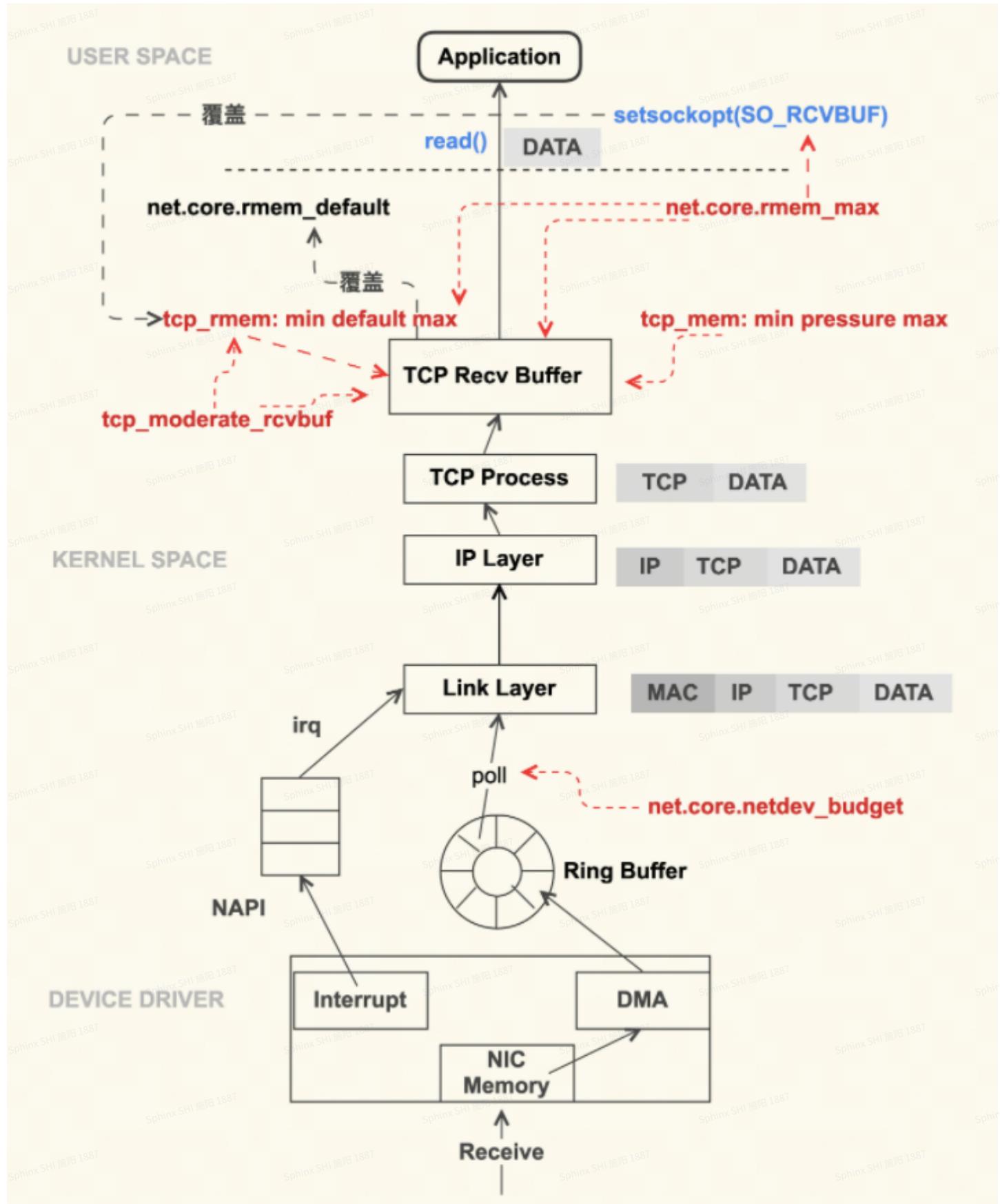
1	\$ nstat -z grep -i drop		
2	TcpExtLockDroppedIcmps	0	0.0
3	TcpExtListenDrops	0	0.0
4	TcpExtTCPBacklogDrop	0	0.0
5	TcpExtPFMemallocDrop	0	0.0
6	TcpExtTCPMinTTLDrop	0	0.0
7	TcpExtTCPDeferAcceptDrop	0	0.0
8	TcpExtTCPReqQFullDrop	0	0.0
9	TcpExtTCP0F0Drop	0	0.0
10	TcpExtTCPZeroWindowDrop	0	0.0

11	TcpExtTCPRecvQDrop	0	0.0
12	MPTcpExtAddAddrDrop	0	0.0
13	MPTcpExtRmAddrDrop	0	0.0

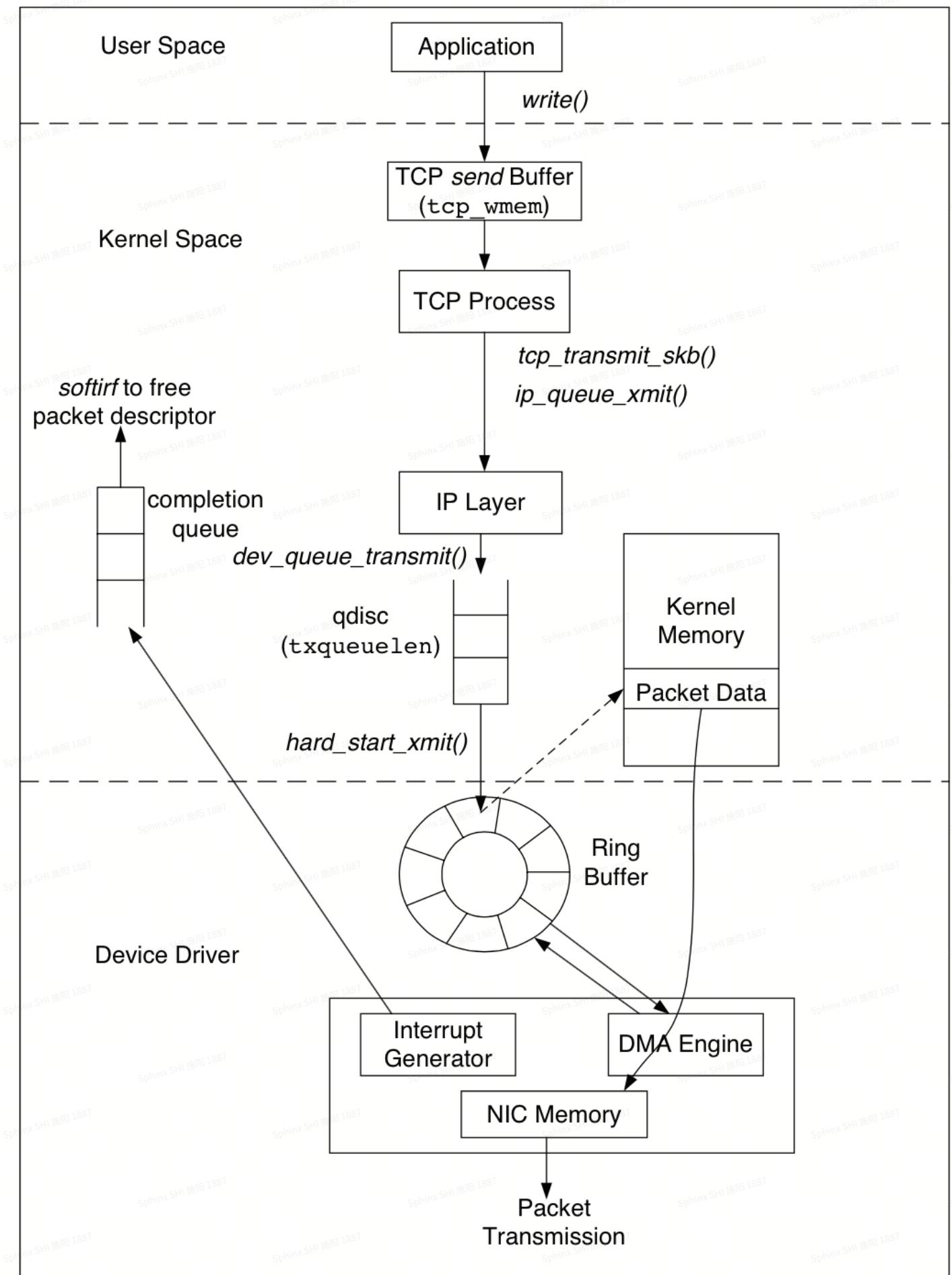
总体流程

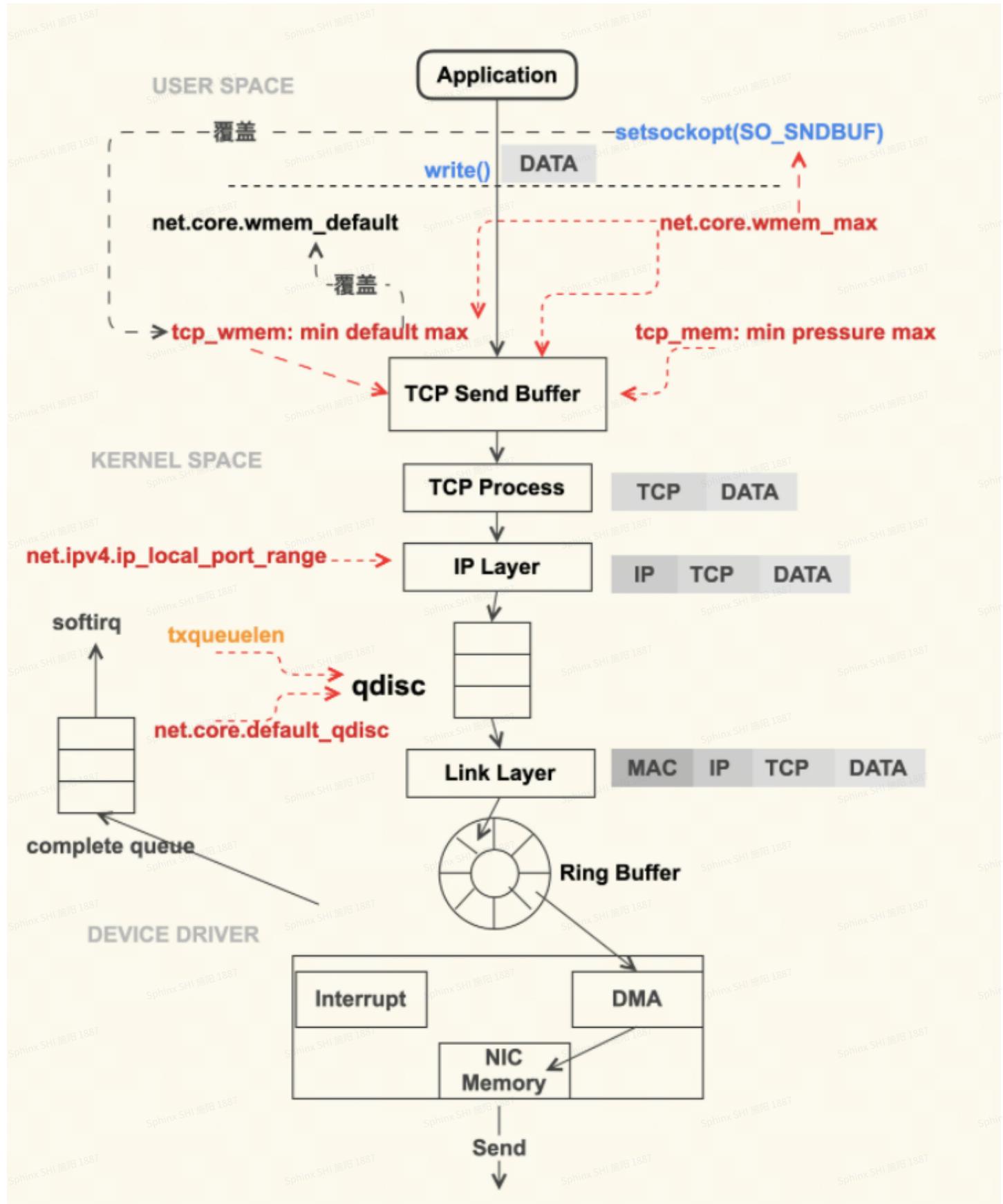
收



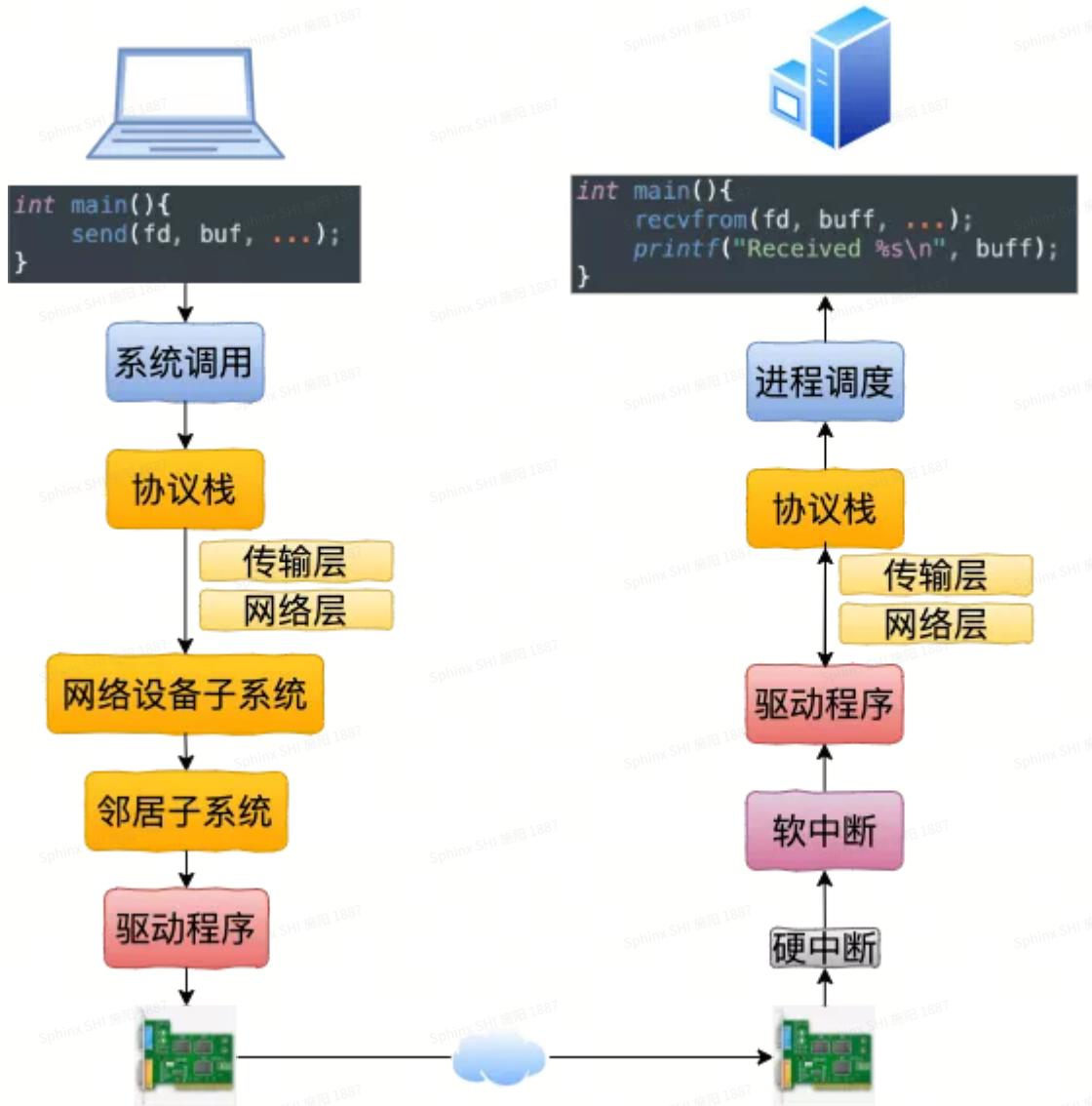


发



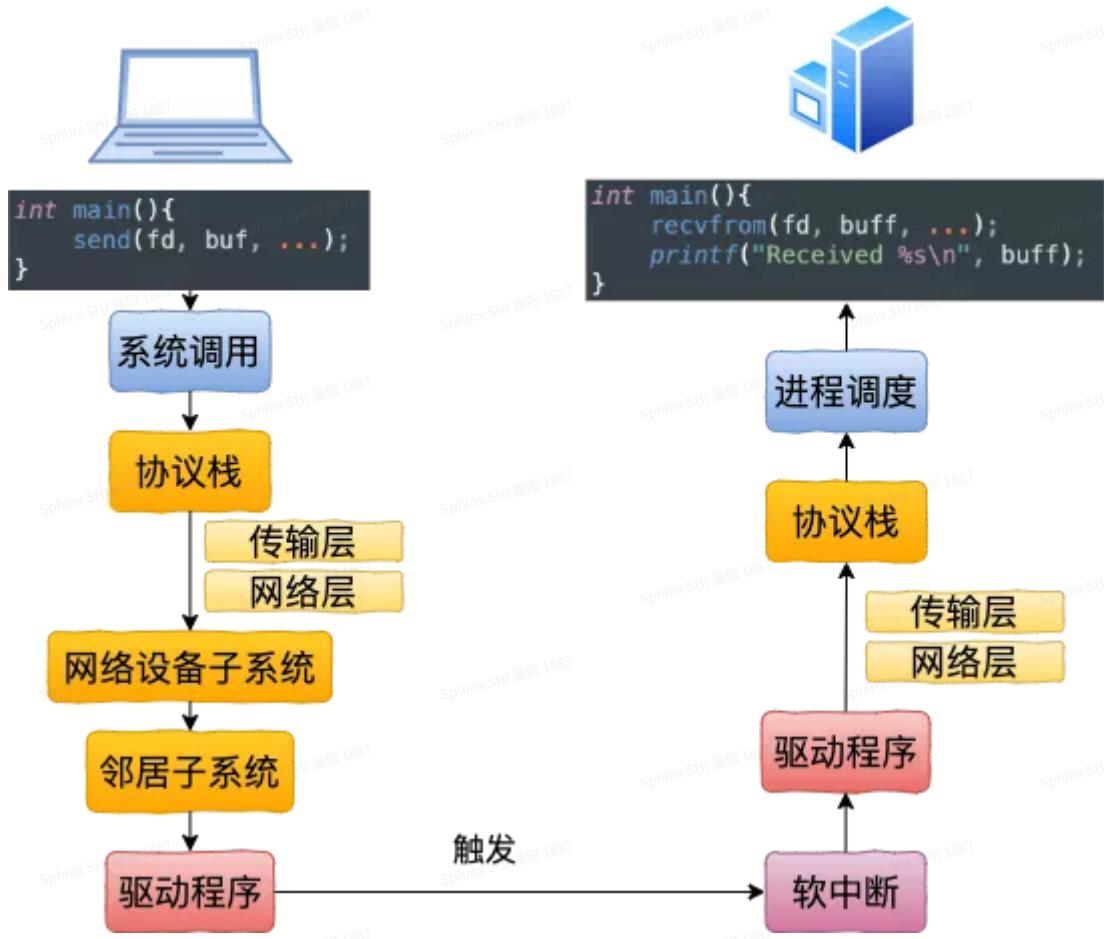


跨机器网络IO



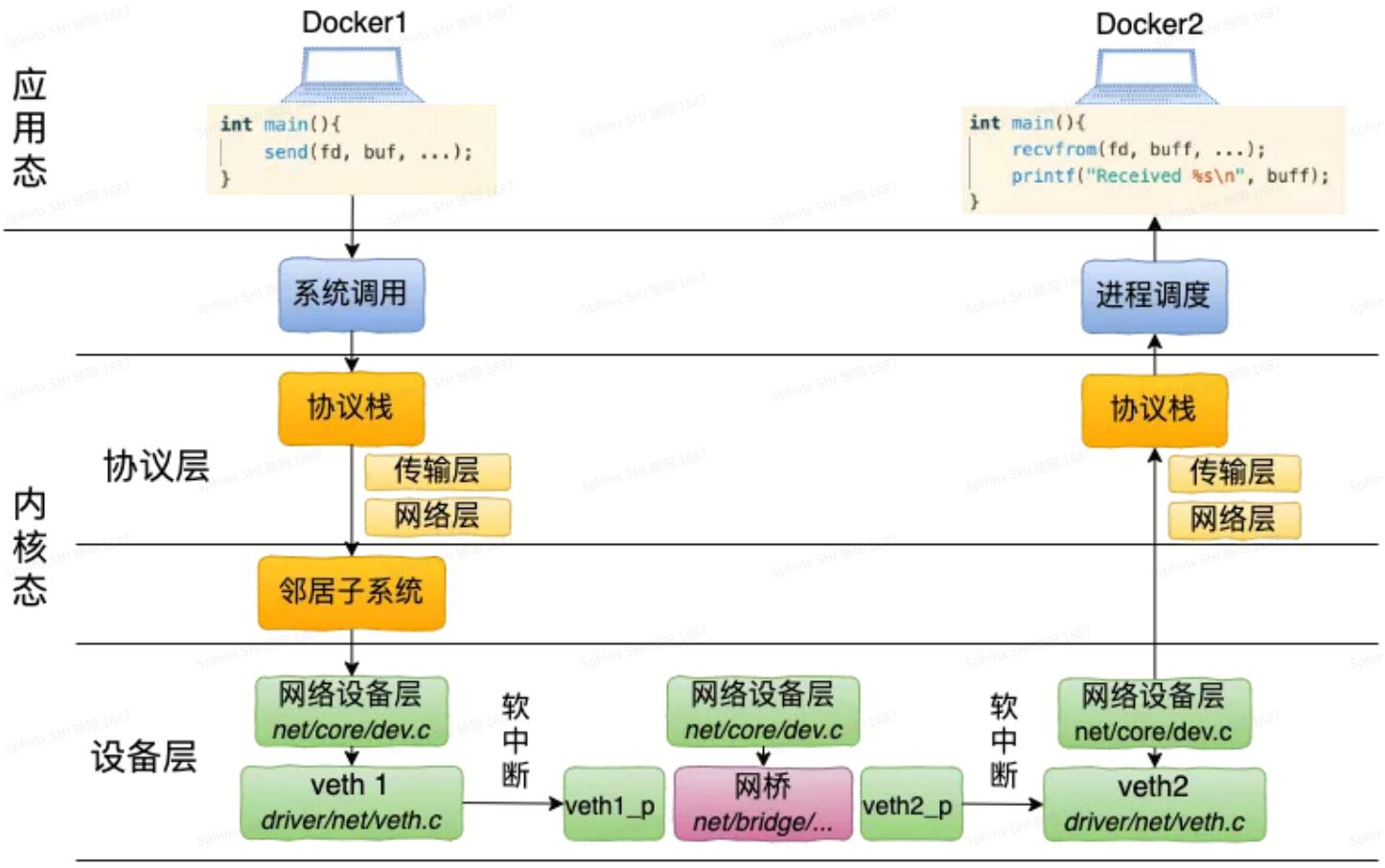
lo 网卡

127.0.0.1(lo)本机网络 IO，无需走到物理网卡，也不用进入RingBuffer驱动队列，但是还是要走内核协议栈，直接把 skb 传给接收协议栈（经过软中断）



总的来说，本机网络 IO 和跨机 IO 比较起来，确实是节约了一些开销。发送数据不需要进 RingBuffer 的驱动队列，直接把 skb 传给接收协议栈（经过软中断）。但是在内核其它组件上，可是一点都没少，系统调用、协议栈（传输层、网络层等）、网络设备子系统、邻居子系统整个走了一个遍。连“驱动”程序都走了（虽然对于回环设备来说只是一个纯软件的虚拟出来的东东）。所以即使是本机网络 IO，也别误以为没啥开销，实际本机访问自己的eth0 ip也是走的lo网卡和访问127.0.0.1是一样的，测试用ab分别走127.0.0.1和eth0压nginx，在nginx进程跑满，ab还没满两者的nginx单核都是7万TPS左右，跨主机压nginx的单核也是7万左右（要调多ab的并发数）。

如果是同一台宿主机走虚拟bridge通信的话（同一宿主机下的不同docker容器通信）：



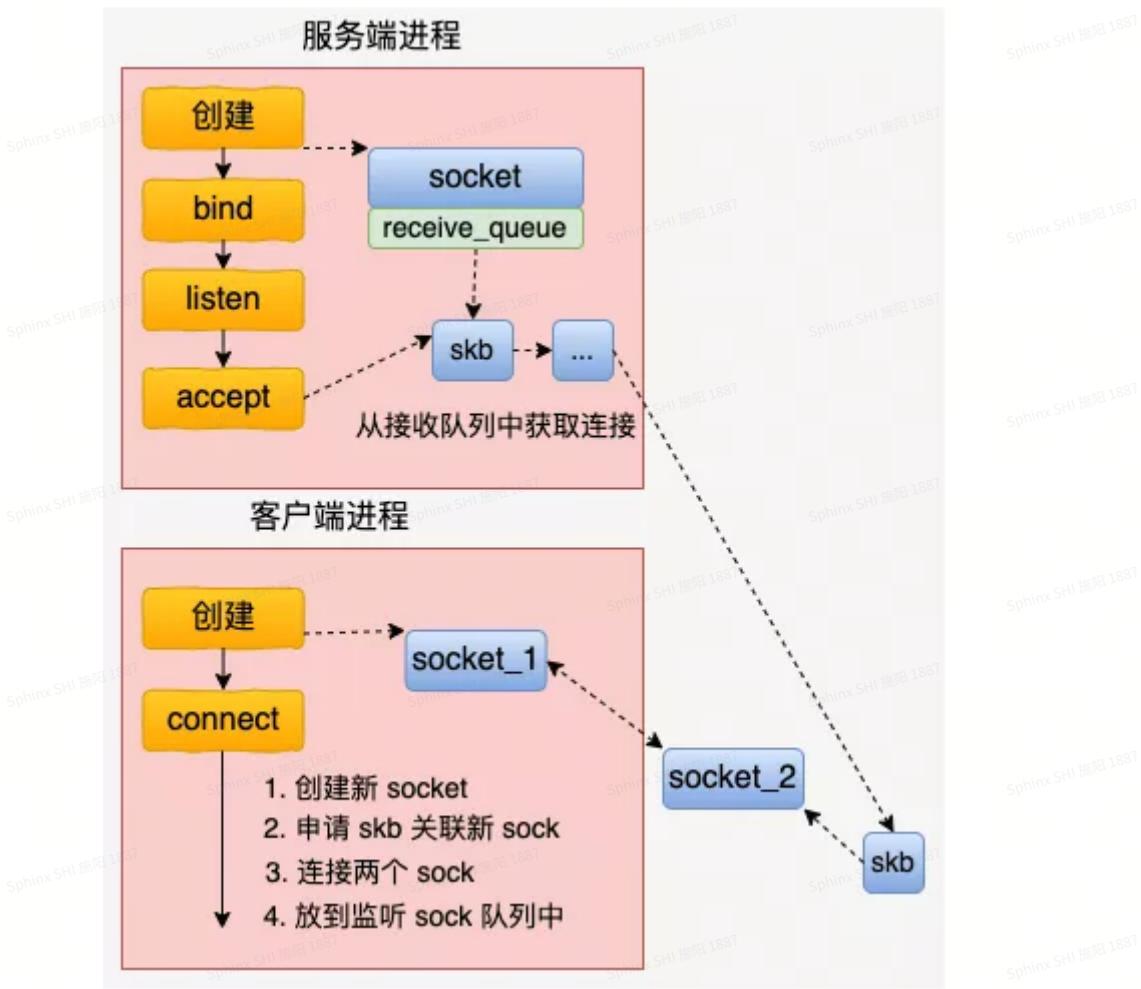
ab 压 nginx单核 (intel 8163 绑核)	
127.0.0.1	Requests per second: 69498.96 [#/sec] (mean) Time per request: 0.086 [ms] (mean)
Eth0	Requests per second: 70261.93 [#/sec] (mean) Time per request: 0.085 [ms] (mean)
跨主机压	Requests per second: 70119.05 [#/sec] (mean) Time per request: 0.143 [ms] (mean)

ab不支持unix domain socket，如果增加ab和nginx之间的时延，QPS急剧下降，但是增加ab的并发数完全可以把QPS拉回去。

Unix Domain Socket

接收connect请求的时候，会申请一个新socket给server端将来使用，和自己的socket建立好连接关系以后，就放到服务器正在监听的socket的接收队列中。

这个时候，服务器端通过accept就能获取到和客户端配好对的新socket了。



主要的连接操作都是在这个函数中完成的。和平常所见的 TCP 连接建立过程，这个连接过程简直太简单了。没有三次握手，也没有全连接队列、半连接队列，更没有啥超时重传。

直接就是将两个 socket 结构体中的指针互相指向对方就行了。就是 `unix_peer(newsk) = sk` 和 `unix_peer(sk) = newsk` 这两句。

代码块

```

1 //file: net/unix/af_unix.c
2 static int unix_stream_connect(struct socket *sock, struct sockaddr *uaddr,
3     int addr_len, int flags)
4 {
5     struct sockaddr_un *sunaddr = (struct sockaddr_un *)uaddr;
6
7     // 1. 为服务器侧申请一个新的 socket 对象
8     newsk = unix_create1(sock_net(sk), NULL);
9
10    // 2. 申请一个 skb，并关联上 newsk
11    skb = sock_wmalloc(newsk, 1, 0, GFP_KERNEL);
12    ...
13
14    // 3. 建立两个 socket 对象之间的连接
15    unix_peer(newsk) = sk;
16    newsk->sk_state = TCP_ESTABLISHED;

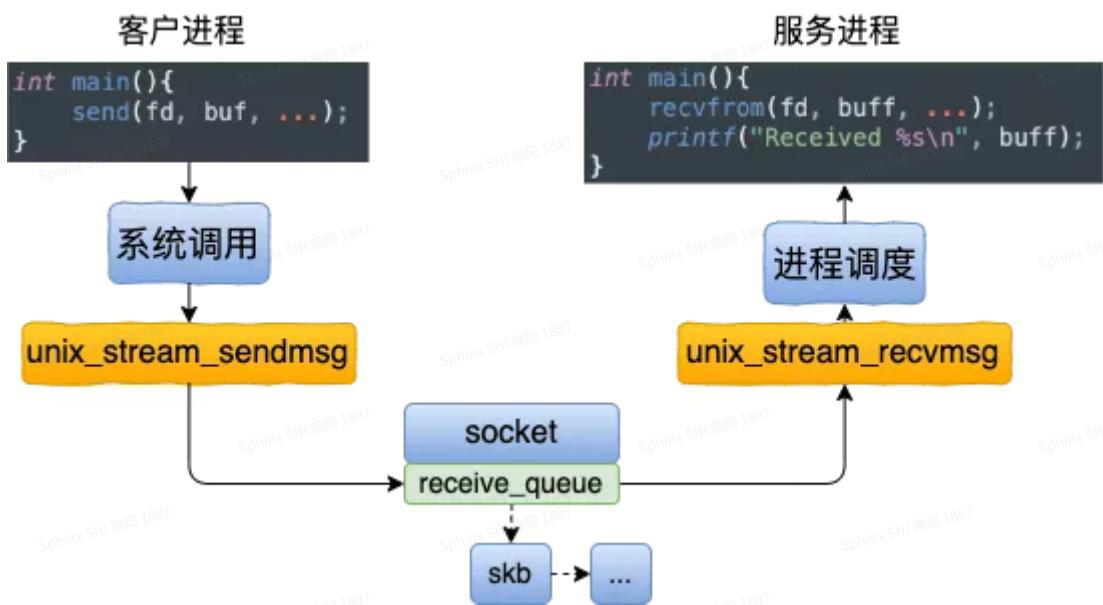
```

```

17     newsk->sk_type = sk->sk_type;
18     ...
19     sk->sk_state = TCP_ESTABLISHED;
20     unix_peer(sk) = newsk;
21
22     // 4. 把连接中的一头 (新 socket) 放到服务器接收队列中
23     __skb_queue_tail(&other->sk_receive_queue, skb);
24 }
25
26 //file: net/unix/af_unix.c
27 #define unix_peer(sk) (unix_sk(sk)->peer)

```

收发包过程和复杂的 TCP 发送接收过程相比，这里的发送逻辑简单到令人发指。申请一块内存 (skb)，把数据拷贝进去。根据 socket 对象找到另一端，直接把 skb 给放到对端的接收队列里了



Unix Domain Socket和127.0.0.1通信相比，如果包的大小是1K以内，那么性能会有一倍以上的提升，包变大后性能的提升相对会小一些。

延迟测试

Case	方法	包大小 (字节)	测试次数	平均延迟 (纳秒)
小包	UDS	100	1000万	2707
	TCP	100	1000万	5690
中包	UDS	1000	1000万	2753
	TCP	1000	1000万	5627
较大包	UDS	100000	1万	24175
	TCP	100000	1万	32683

吞吐测试

Case	方法	包大小(字节)	测试次数	平均每秒消息数	平均带宽(Mb/s)
小包	UDS	100	1000万	1068321	854
	TCP	100	1000万	483059	386
中包	UDS	1000	1000万	918874	7350
	TCP	1000	1000万	411687	3293
较大包	UDS	30000	100万	150494	36118
	TCP	30000	100万	123538	29649

再来一个整体流转矢量图：

