IPADS INSTITUTE OF PARALLEL AND DISTRIBUTED SYSTEMS

上海交通大学 SHANGHAI JIAO TONG UNIVERSITY

网络

陈海波 / 夏虞斌

上海交通大学并行与分布式系统研究所

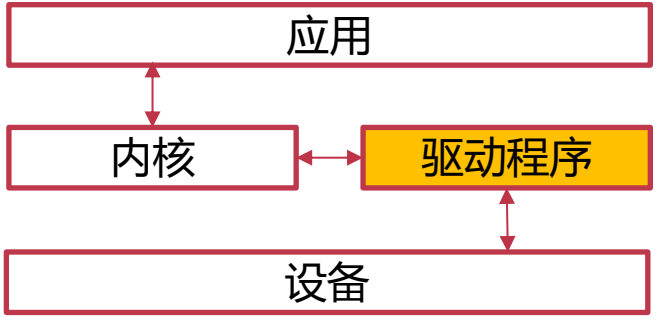
https://ipads.se.sjtu.edu.cn

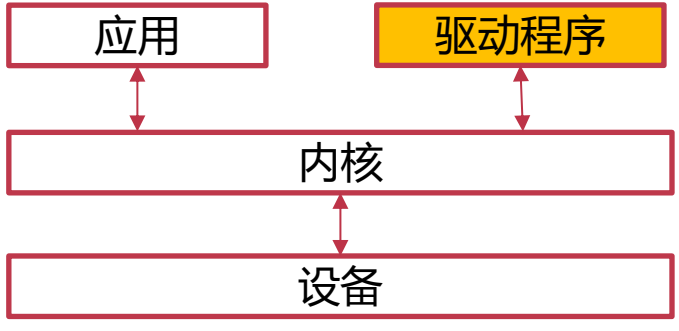
版权声明

- 本内容版权归**上海交通大学并行与分布式系统研究所**所有
- 使用者可以将全部或部分本内容免费用于非商业用途
- 使用者在使用全部或部分本内容时请注明来源：
  - 内容来自：上海交通大学并行与分布式系统研究所+材料名字
- 对于不遵守此声明或者其他违法使用本内容者，将依法保留追究权
- 本内容的发布采用 Creative Commons Attribution 4.0 License
  - 完整文本：https://creativecommons.org/licenses/by/4.0/legalcode

回顾：宏内核vs微内核的驱动

- **宏内核**
  - 驱动在内核态
  - 优势：性能更好
  - 劣势：容错性差
- **微内核**
  - 驱动在用户态
  - 优势：可靠性好
  - 劣势：性能开销（IPC）



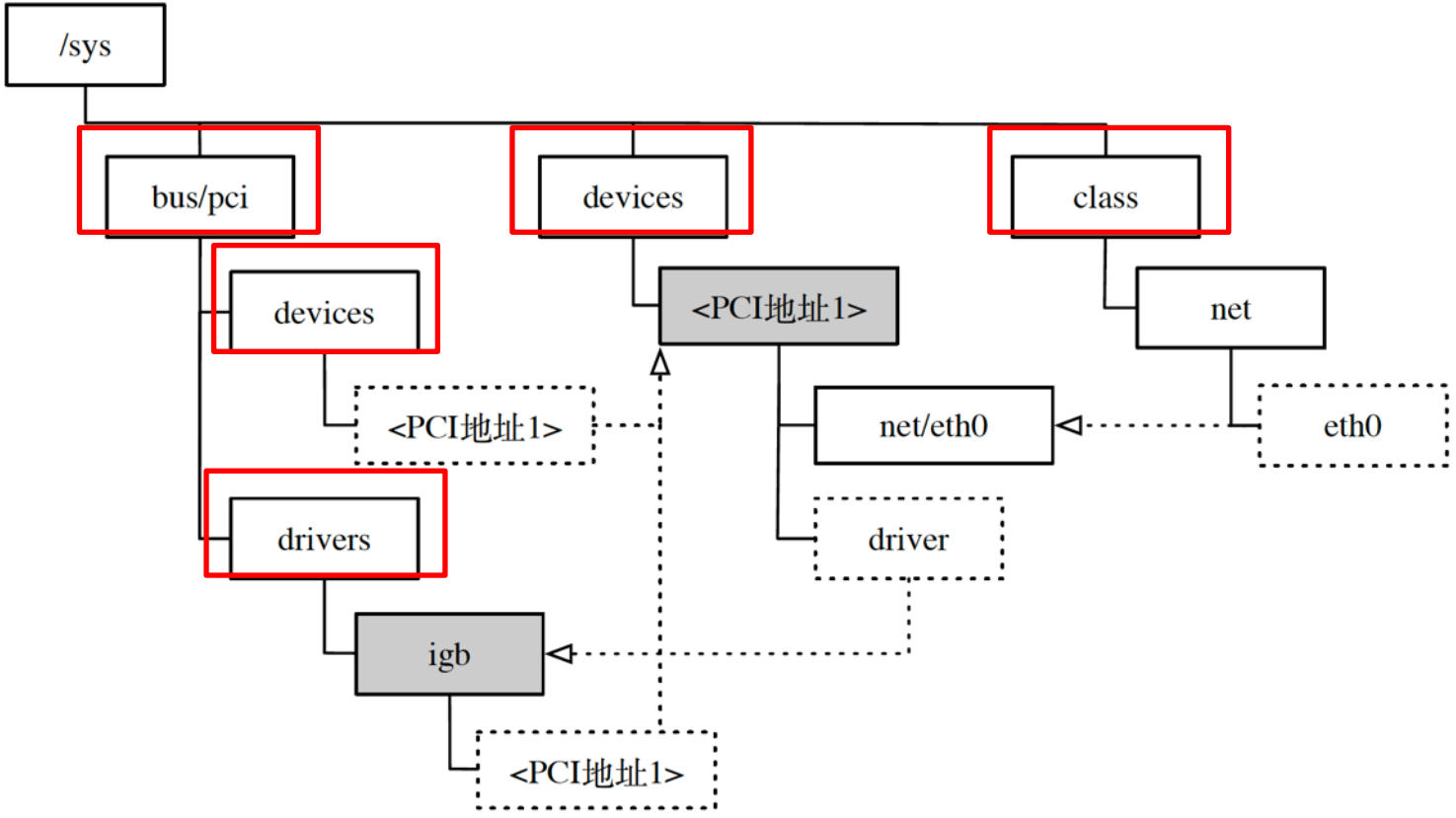


回顾：设备驱动模型

- **设备驱动模型**
  - 提供标准化的数据结构和接口
  - 将驱动开发简化为对数据结构的填充和实现
  - 方便操作系统统一组织和管理设备

- **Linux上下半部**
  - 上半部：尽量快，提高对外设的响应能力
  - 下半部：将中断的处理推迟完成

回顾：sysfs

- 
/sys
bus/pci
devices
class
devices
<PCI地址1>
net
<PCI地址1>
net/eth0
eth0
drivers
driver
igb
<PCI地址1>

中断子系统

案例：LINUX的上下半部

Top Half：马上做

- **最小化公共例程：**
  - 保存寄存器、屏蔽中断
  - 恢复寄存器，返回现场
- **最重要：调用合适的由硬件驱动提供的中断处理handler**
- **因为中断被屏蔽，所以不要做太多事情（时间、空间）**
- **将请求放入队列（或设置flag），将其他处理推迟到bottom half**

Top Half：找到handler

思考：
为什么要共享IRQ？

- 现代处理器中，多个I/O设备共享一个IRQ和中断向量
- 多个ISR (interrupt service routines)可以绑定在同一个向量上
- 调用每个设备对应的IRQ的ISR

Bottom Half：延迟完成

- **提供可以推迟完成任务的机制**
  - softirqs
  - tasklets（建立在softirqs之上)
  - 工作队列
  - 内核线程
- **这些机制都可以被中断**

软中断（Softirqs）

- 静态分配：在内核编译时期确定
- 数量有限：

Priority	*Type*
0	High-priority tasklets
1	Timer interrupts
2	Network transmission
3	Network reception
4	Block devices
5	Regular tasklets

Softirq特点

- **执行时间点：**
  - 中断之后（上半部之后）
  - 系统调用或是异常发生之后
  - 调度器显式执行ksoftirqd
- **并发：**
  - 可以在多核上同时执行
  - 必须是可重入的
  - 或根据需要加锁
- **可中断：Softirq运行时可再被中断抢占**

Softirq Rescheduling

- **软中断要求能被重调度**
  - 在处理软中断A时，能切换至软中断B（挂起A唤醒B）
- **问题：在处理软中断A时，软中断产生了B，怎么办？**
  - 不处理→B响应被延迟
  - 总是处理→如果软中断很长→用户程序被饿死？ **活锁！**
- **方案：配额（quota）+ `ksoftirqd`**
  - Softirq调度器每次只运行有限数量的请求
  - 剩余请求有内核线程`ksoftirqd`代为执行，和用户进程抢CPU
  - `ksoftirqd`和用户进程都被调度器调度

Tasklet

- **问题：Softirq是静态的**
- **方案：引入Tasklet**
  - 基于Softirqs，但可以被动态创建和销毁！
  - 同一时期，同种类型的Tasklet一次只能运行一个
  - 不同类型的Tasklet可以同时运行在不同CPU上
- **缺点：**
  - 和Softirq一样缺乏进程上下文，无法睡眠！

Tasklet的优势

- **可动态分配，数量不限**
- **直接运行在调度它的CPU上（缓存亲和性）**
- **执行期间不能被其它下半部抢占**
  - 不存在重入的问题
  - 无需加锁
- **曾是最受原因的延迟处理机制**

Tasklet的问题

- **难以正确实现**
  - 要防止休眠代码
- **任务不可抢占性（仍可被中断）**
  - 比其他任务的优先级都高，影响任务实时性
  - 导致不可控的延迟
- **Linux社区一直在讨论是否要移除Tasklet**

工作队列（Work Queues）

- **Softirq和Tasklet使用中断上下文**
- **工作队列使用进程上下文**
  - 可以睡眠！
- **方式：**
  - 在内核空间维护FIFO队列， *workqueue*内核进程不断轮询队列
  - 中断负责enqueue(fn, args)， *workqueue*负责dequeue并执行fn(args)
- **特点：**
  - 只在内核空间，不和任何用户进程关联，没有跨模式切换和数据拷贝

内核线程（Kernel Threads）

- **始终运行在内核态**
  - 和工作队列一样，没有用户空间上下文
- **中断线程化*（*threaded interrupt handlers*）**
  - Linux 2.6.30引入
  - 想要取代Tasklet和Workqueue
  - 每个中断线程都有自己的上下文

* [LWN] Moving interrupts to threads, https://lwn.net/Articles/302043/

总结

特点	ISR	SoftIRQ	Tasklet	WorkQueue	KThread
禁用所有中断？	Briefly	No	No	No	No
禁用相同优先级的中断？	Yes	Yes	No	No	No
比常规任务优先级更高？	Yes	Yes*	Yes*	No	No
在相同处理器上运行？	N/A	Yes	Yes	Yes	Maybe
允许在同一CPU上有多个实例同时运行？	No	No	No	Yes	Yes
允许在多个CPU上运行同时多个相同实例？	Yes	Yes	No	Yes	Yes
完整的模式切换？	No	No	No	Yes	Yes
能否睡眠（拥有自己的内核栈）？	No	No	No	Yes	Yes
能否访问用户空间？	No	No	No	No	No

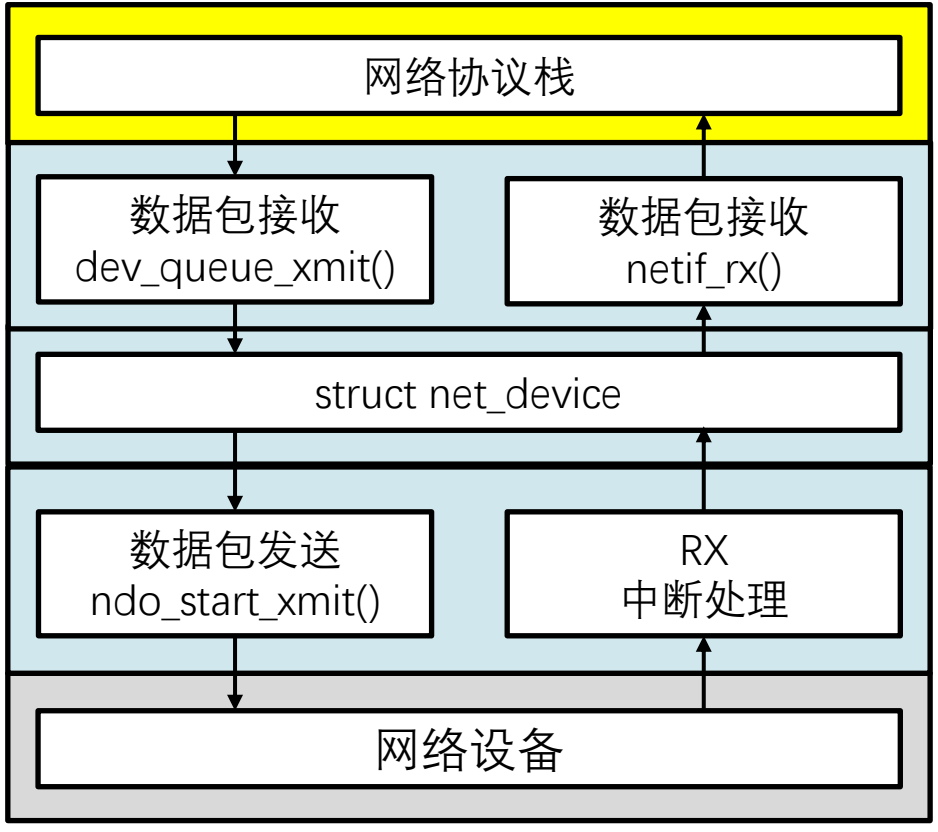
* 可以由 ksoftirqd 调度

操作系统的网络

网络协议栈的分层模型

	应用层	游戏进程或聊天进程
L4	传输层	TCP或UDP协议：端口
L3	网络层	IP协议：主机地址
L2	数据链路层	网络设备驱动
	物理层	网络设备

Linux网络驱动模型



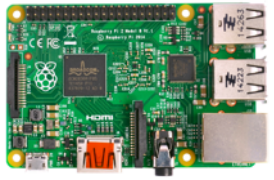
网络协议接口层

网络设备接口层

设备驱动层

物理层

网卡硬中断（ISR）



树莓派3

```
$ cat /proc/interrupts
           CPU0       CPU1       CPU2       CPU3
 17:          0          0          0          0     GICv2  29 Level     arch_timer
 18:    7331554    2731032     433991     492919     GICv2  30 Level     arch_timer
 23:      26740          0          0          0     GICv2 114 Level     DMA IRQ
 31:     757858          0          0          0     GICv2  65 Level     fe00b880.mailbox
 34:       6556          0          0          0     GICv2 153 Level     uart-pl011
 36:          0          0          0          0     GICv2 169 Level     brcmstb_thermal
 37:    8457672          0          0          0     GICv2 158 Level     mmc1, mmc0
 43:          0          0          0          0     GICv2 106 Level     v3d
 45:    7567287          0          0          0     GICv2 189 Level     eth0
 52:         51          0          0          0     GICv2  66 Level     VCHIQ doorbell
 53:          0          0          0          0     GICv2 175 Level     PCIe PME, aerdrv
 54:         40          0          0          0  Brcm_MSI 524288 Edge      xhci_hcd
```

网卡软中断（softirq）

树莓派3

```
$ cat /proc/softirqs
```

	CPU0	CPU1	CPU2	CPU3
HI:	2	0	0	0
TIMER:	4709143	1000453	238535	156693
NET_TX:	12764	272	293	196
NET_RX:	650451	4930	7150	5162
BLOCK:	0	0	0	0
IRQ_POLL:	0	0	0	0
TASKLET:	6775576	36	24	33
SCHED:	4719393	1043269	255401	165523
HRTIMER:	0	0	0	0
RCU:	2878697	423063	251156	170016

网卡收发的情况

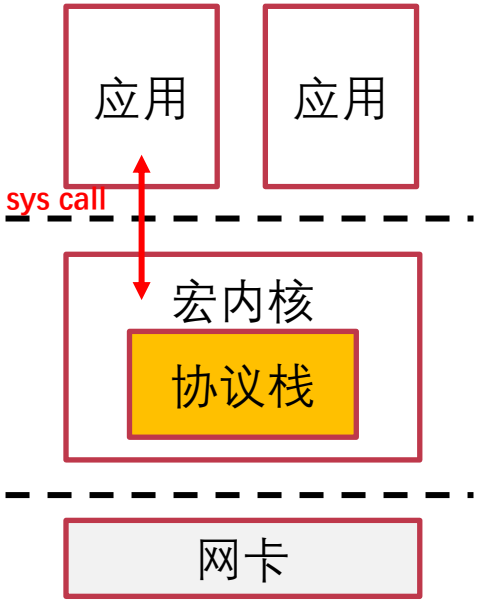
树莓派3

```
$ ifconfig
eth0: flags=4099<UP,BROADCAST,MULTICAST>  mtu 1500
        ether dc:a6:32:4b:c4:00  txqueuelen 1000  (Ethernet)

wlan0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.10.194  netmask 255.255.0.0  broadcast 192.168.255.255
        inet6 fe80::aa72:beb8:1888:e82e  prefixlen 64  scopeid 0x20<link>
        ether dc:a6:32:4b:c4:01  txqueuelen 1000  (Ethernet)
        RX packets 655811  bytes 164726673 (157.0 MiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 21714  bytes 2496958 (2.3 MiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

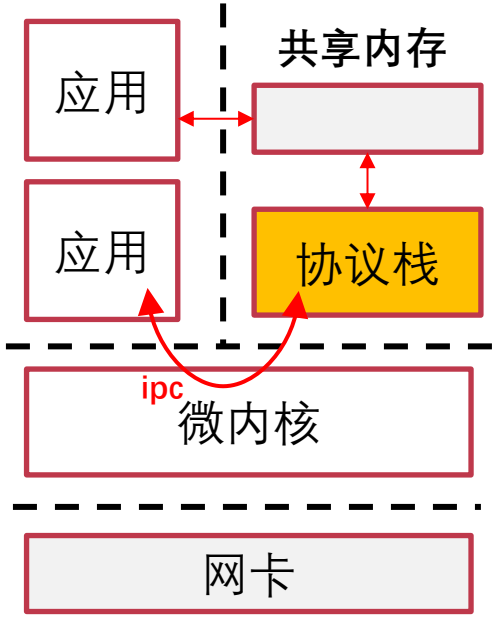
中断合并

- **Interrupt coalescing**
  - 当外设中断次数累计到一定阈值时，再向CPU发送中断
  - 或者到某个timeout，向CPU发送中断
- **可避免“中断风暴”**
- **更少的中断次数意味着：**
  - 更高的吞吐量
  - 但也增加了中断响应的延迟

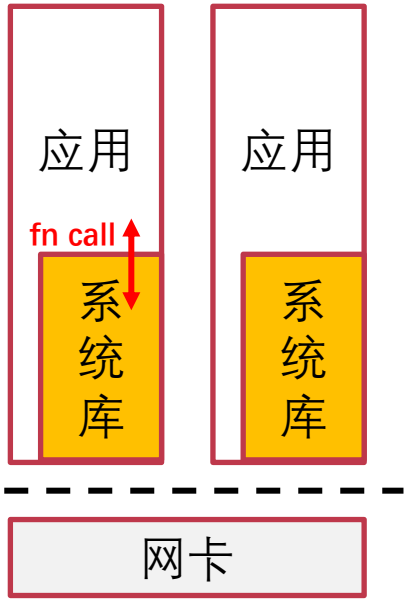
架构对比



宏内核系统



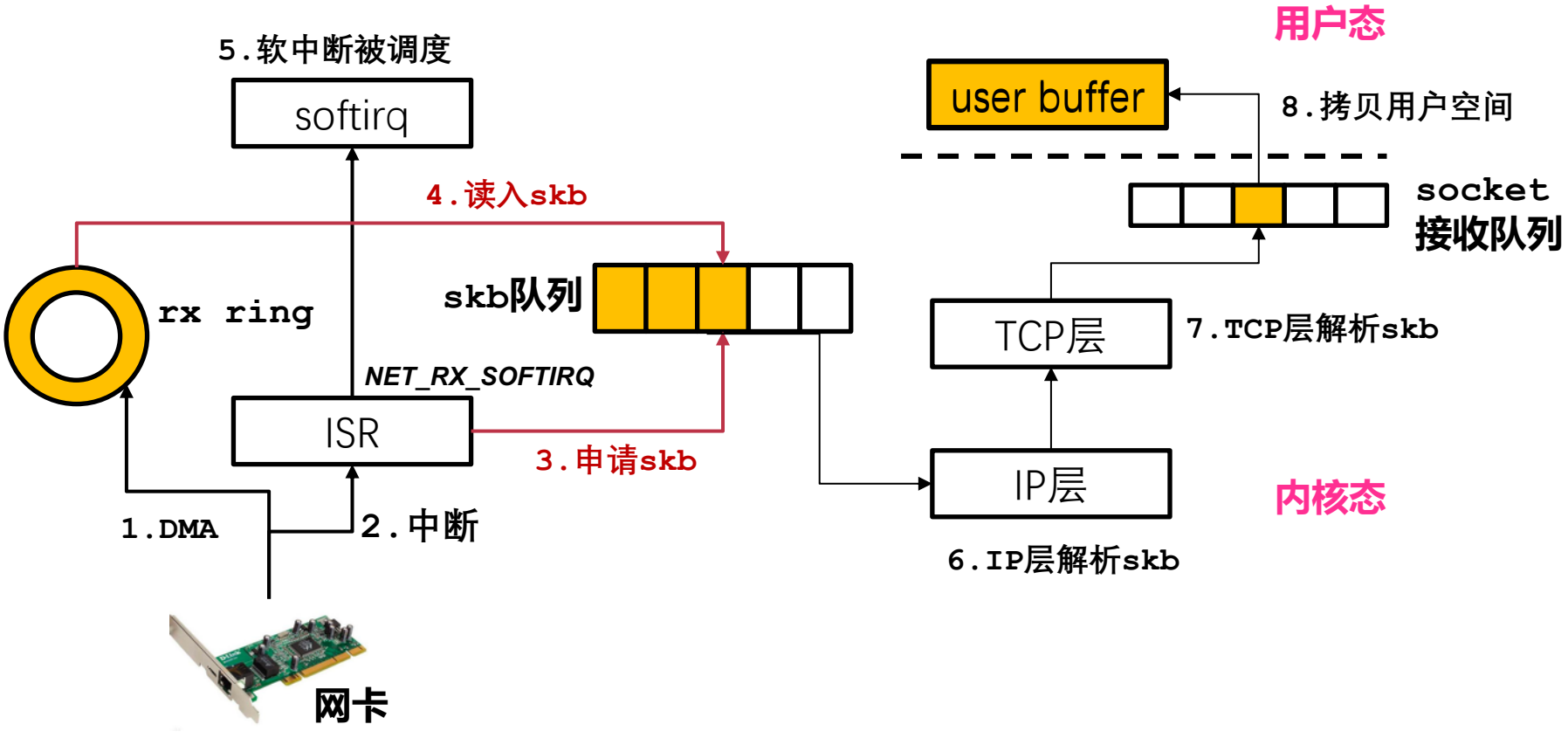
微内核系统



库操作系统

案例：LINUX的收包过程

Linux收包过程


用户态
5.软中断被调度
softirq
user buffer
8.拷贝用户空间
4.读入skb
socket
接收队列
rx ring
skb队列
NET_RX_SOFTIRQ
TCP层
7.TCP层解析skb
ISR
3.申请skb
IP层
内核态
1.DMA
2.中断
6.IP层解析skb
网卡

收包：数据链路层

- **网卡收到到数据包（以太网帧）：**
  - DMA 将数据帧传送至内核内存中的rx_ring
  - 网卡中断被触发
- **CPU收到网卡中断，调用网卡ISR（上半部）：**
  - 分配 *sk_buff (skb)* 数据结构，负责管理rx_ring中的数据包
  - 将skb包入队（*input_pkt_queue*）
- **上半部发出一个软中断（*NET_RX_SOFTIRQ*）：**
  - 通知内核处理skb包

收包：数据链路层（2）

- **进入软中断处理流程（下半部）：**
  - 把 *input_pkt_queue* 的skb移动到 *process_queue* 处理队列中
  - 根据报文类型(ARP或是IP)，把报文递交给对应协议进行处理
  - 调用网络层协议的的handler处理skb包
- **移动skb：操作指针**
- **如果接收队列已满（input_pkt_queue）：**
  - 丢弃后续数据：可能发生**活锁！**

收包：网络层

- **IP 层的入口函数 ip_rcv()：**
  - 检查是否为IP包
  - 检查IP版本号
  - 对完整性（checksum）和长度进行检查
- **ip_rcv()结束调用ip_router_input()，进行路由处理**
  - 查找路由
  - 决定该数据包（报文）是发到本机，还是被转发，或是被丢弃

收包：传输层

- **传输层的处理入口 tcp_v4_rcv()**
  - 对 TCP header 进行检查
- **调用 _tcp_v4_lookup，查找该数据包对应的open socket**
  - 如果找不到，该数据包被丢弃
  - 否则检查 socket 和 connection 的状态
- **socket 和 connection 正常**
  - 调用 tcp_prequeue() 使tcp载荷从内核进入用户空间，放进 socket 接收队列

收包：应用层

- **socket 被唤醒，调用 system call，并最终调用 tcp_recvmsg()，从 socket 接收队列 中获取数据**
- **用户态调用 read 或者 recvfrom ，转化为 sys_recvfrom 调用**
  - 对 TCP 来说，调用 tcp_recvmsg()：该函数从socket buffer 中拷贝数据到user buffer

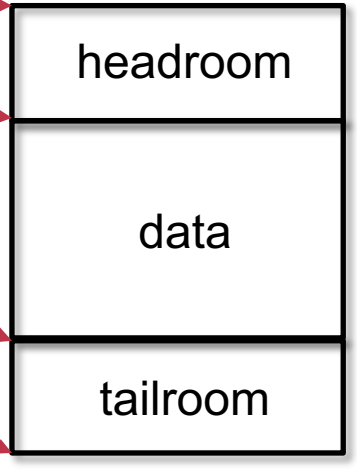
网络包的管理

- **要求高效地处理分层**
  - 发包时需要不断添加新的头部，收包则相反
- **避免连续存放网络包**
  - 移动过程中数据拷贝会有很大开销
- **Linux数据结构：sk_buff（简称skb）**
  - 让分层的处理变得高效：零拷贝
  - 快速申请和释放内存：防止内存碎片

sk_buff

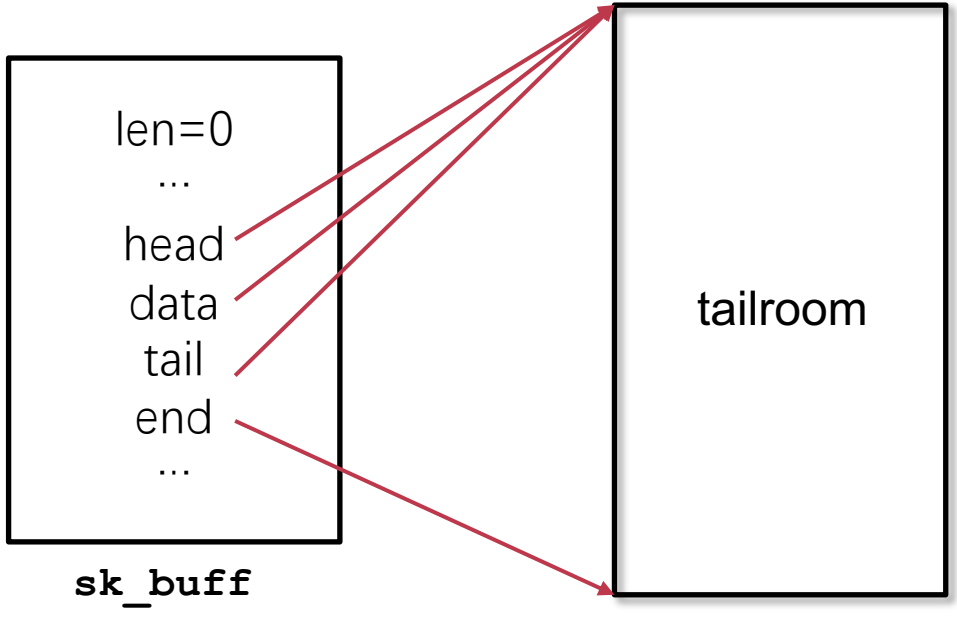
```
struct sk_buff {
    /* These two members must be first. */
    struct sk_buff     * next;
    struct sk_buff     * prev;

    // 真正指向的数据buffer
    struct sock  *sk;

    // 缓冲区的头部
    unsigned char *head;
    // 实际数据的头部
    unsigned char *data;
    // 实际数据的尾部
    unsigned char *tail;
    // 缓冲区的尾部
    unsigned char *end;
};
```

- **sk_buff本身不存储报文**
  - 通过指针指向真正的报文内存空间
- **在各层传递时**
  - 只需调整指针相应位置即可



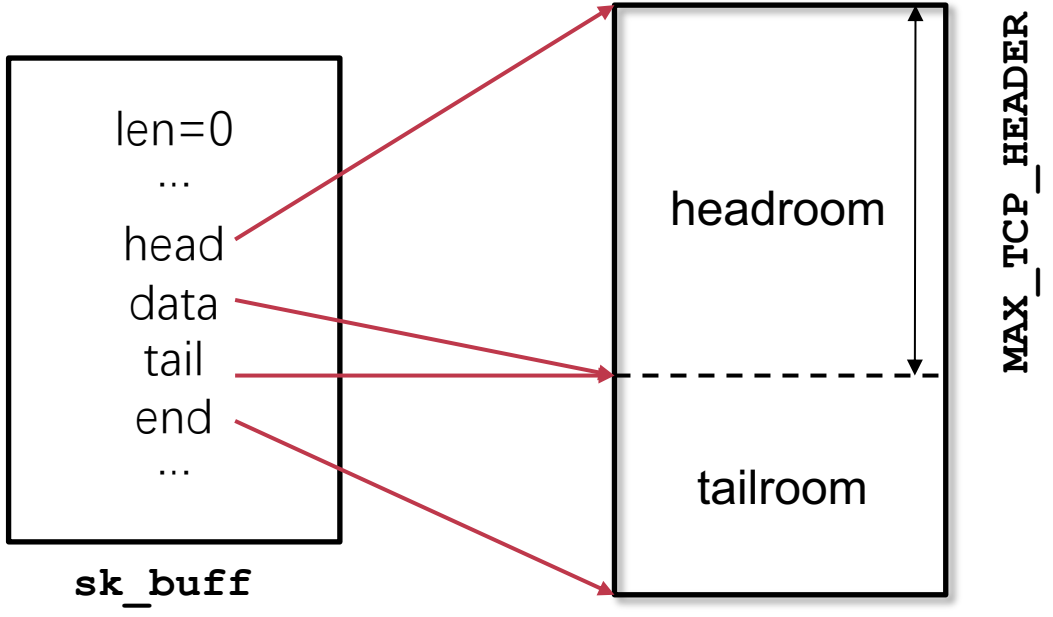
例子：发送数据包

- Step1: TCP层发数据时，首先用alloc_skb申请缓冲区



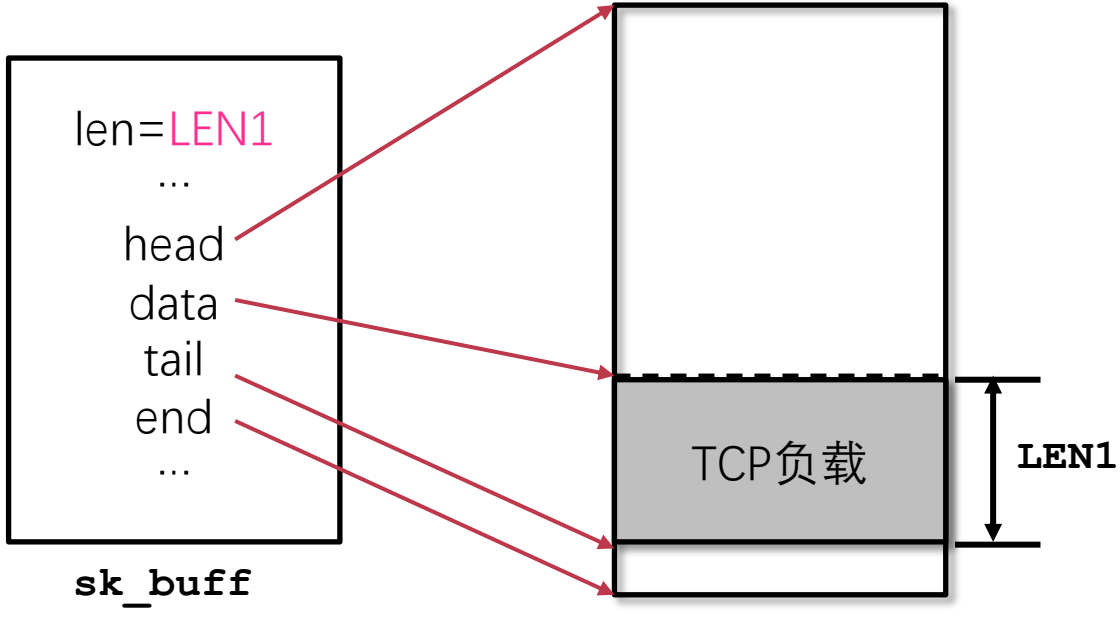
例子：发送数据包

- Step2: TCP用skb_reserve来保留足量空间存储所有头部



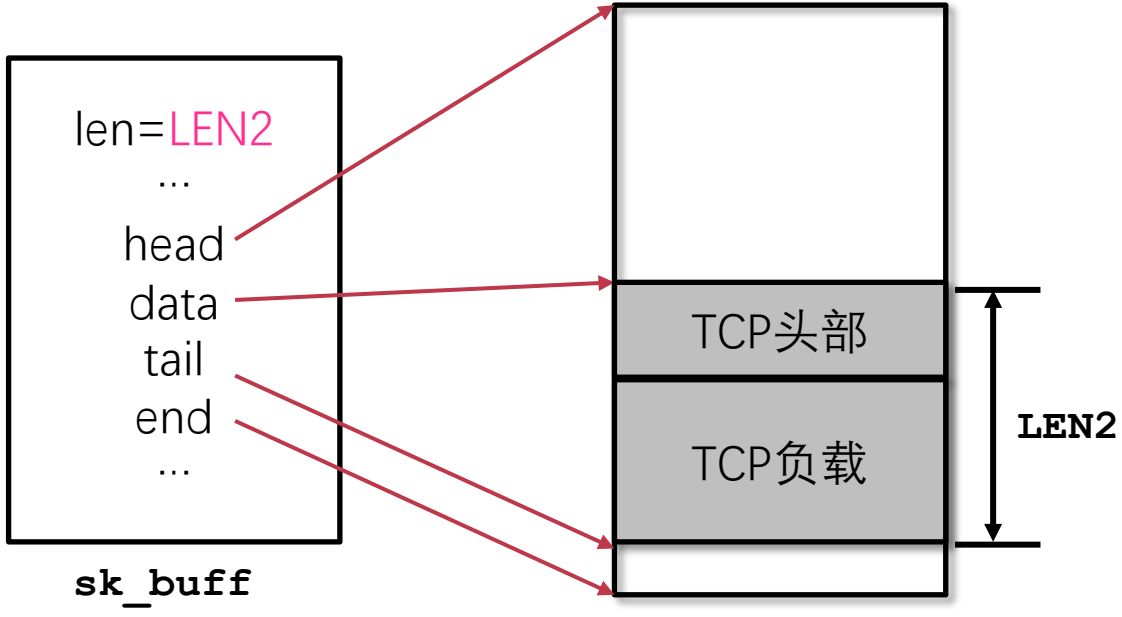
例子：发送数据包

- Step3: TCP层填入TCP负载（应用层数据）



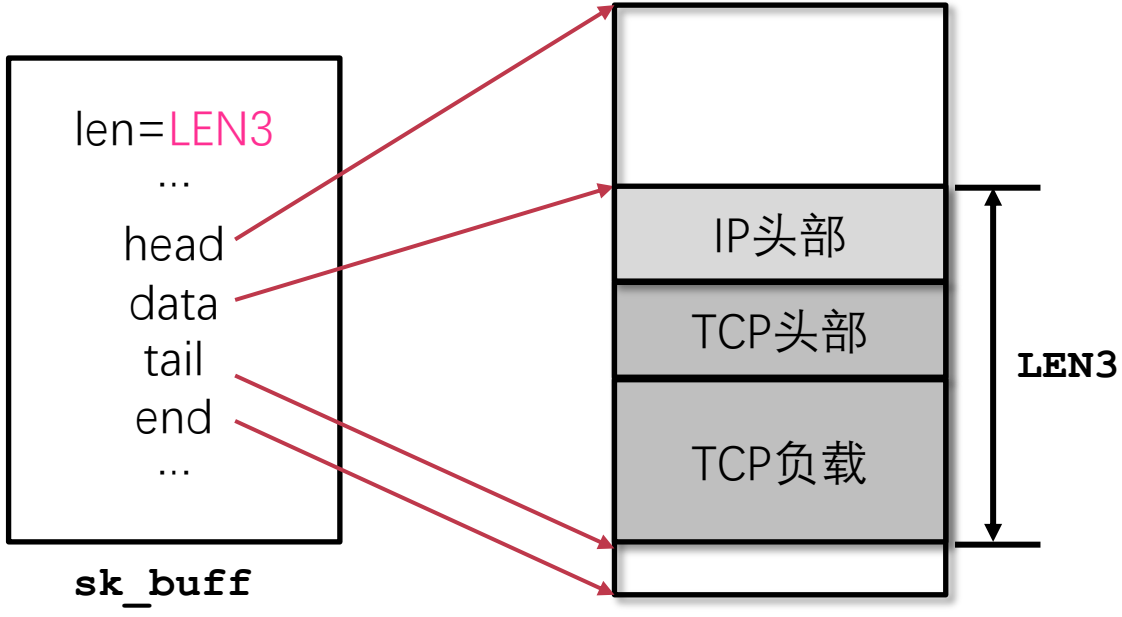
例子：发送数据包

- Step4: TCP层填入TCP头部



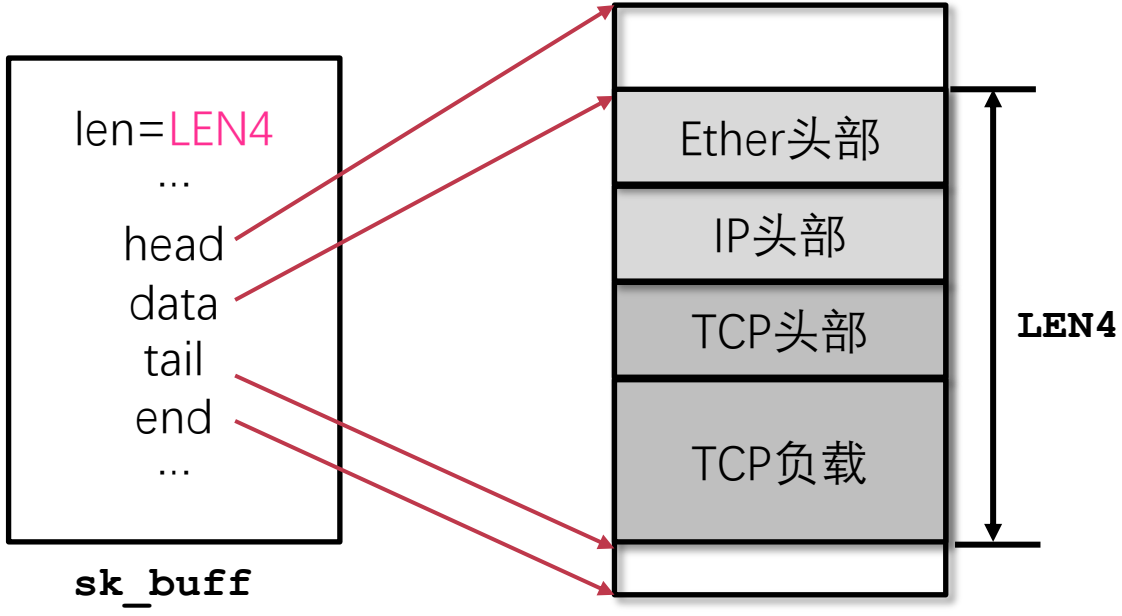
例子：发送数据包

- Step5: 传给IP层，并添加IP头部



例子：发送数据包

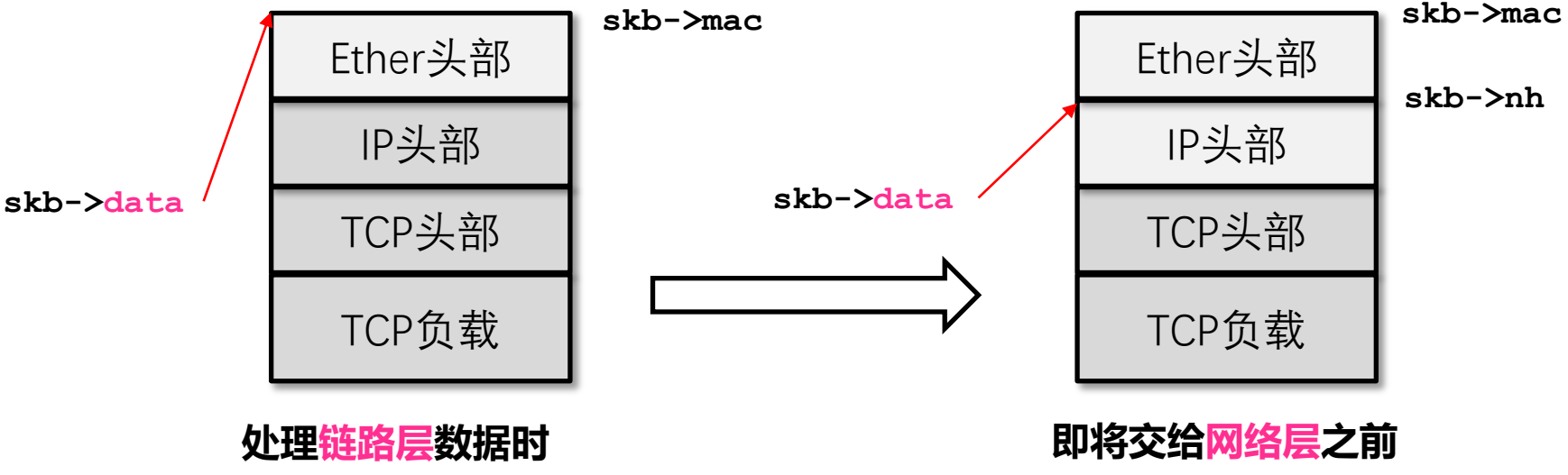
- Step6: 传给链路层，并添加以太网头部



skb的协议栈头部

```
struct sk_buff {
     ... ... ...
     union { ... } h;    // <--- 传输层头部
     union { ... } nh;   // <--- 网络层头部
     union { ... } mac;  // <--- 数据链路层头部
     ... ... ...
};
```

- **当前层处理skb时：**
  - 同时负责将下一层头部指针初始化好，并移动data指针



skb control buffer

- char cb[40];
- 用于每层维护私有的信息

```
struct tcp_skb_cb {
    __u32       seq;        /* Starting sequence number */
    __u32       end_seq;    /* SEQ + FIN + SYN + datalen    */
    ... ... ...
    __u8        tcp_flags;  /* TCP header flags. (tcp[13])  */
    __u8        sacked;     /* State flags for SACK.    */
    __u8        txstamp_ack:1,  /* Record TX timestamp for ack? */
    __u32       ack_seq;    /* Sequence number ACK'd    */
    ... ... ...
    };
};
```

skb的组织

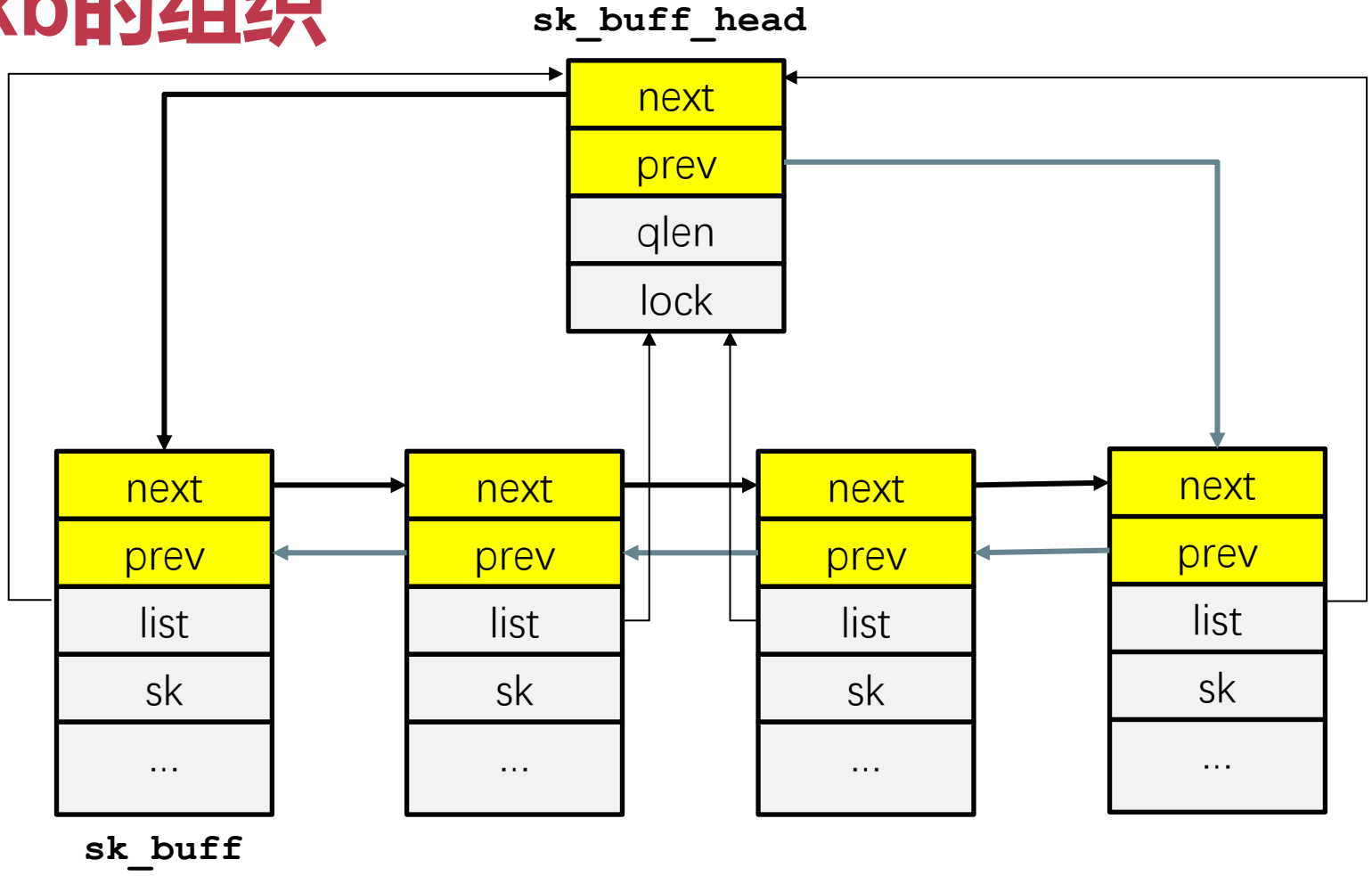
- **用双向链表对sk_buff进行管理：**

```
struct sk_buff_head {
    /* These two members must be first. */
    struct sk_buff    * next;
    struct sk_buff    * prev;
    _ _u32          qlen; // 链表的元素长度
    spinlock_t      lock; // 添加元素时必须持有该锁
};
```

- **Linux NAPI的批处理 ***
  - 让网卡中断处理的下半部softirq积累足量的skb
  - NAPI周期性轮询，并一次性处理完所有skb（batching）
  - *netif_receive_skb_list()*

*** Batch processing of network packets, LWN, https://lwn.net/Articles/763056/**

skb的组织


sk_buff_head
next
prev
qlen
lock
next
prev
list
sk
...
next
prev
list
sk
...
next
prev
list
sk
...
next
prev
list
sk
...
sk_buff

skb操作函数

- **分配：struct sk_buff *alloc_skb(unsigned int size,int gfp_mask)**
  - GFP_ATOMIC：分配过程不能被中断，用于中断上下文中分配内存
  - GFP_KERNEL：分配过程可以被中断，分配请求被放到等待队列中
- **浅拷贝：struct sk_buff *skb_clone(struct sk_buff *skb, int gfp_mask)**
  - 克隆出新的sk_buff控制结构，指向同一报文（用于tcpdump等抓包工具）
- **深拷贝：struct sk_buff *skb_copy(struct sk_buff *skb, int gfp_mask)**
  - 同时复制sk_buff以及指向的报文（用于修改报文，如NAT地址转换）
- **释放：void kfree_skb(struct sk_buff *skb)**
- **使用“引用计数”来管理sk_buff**

GFP_ATOMIC

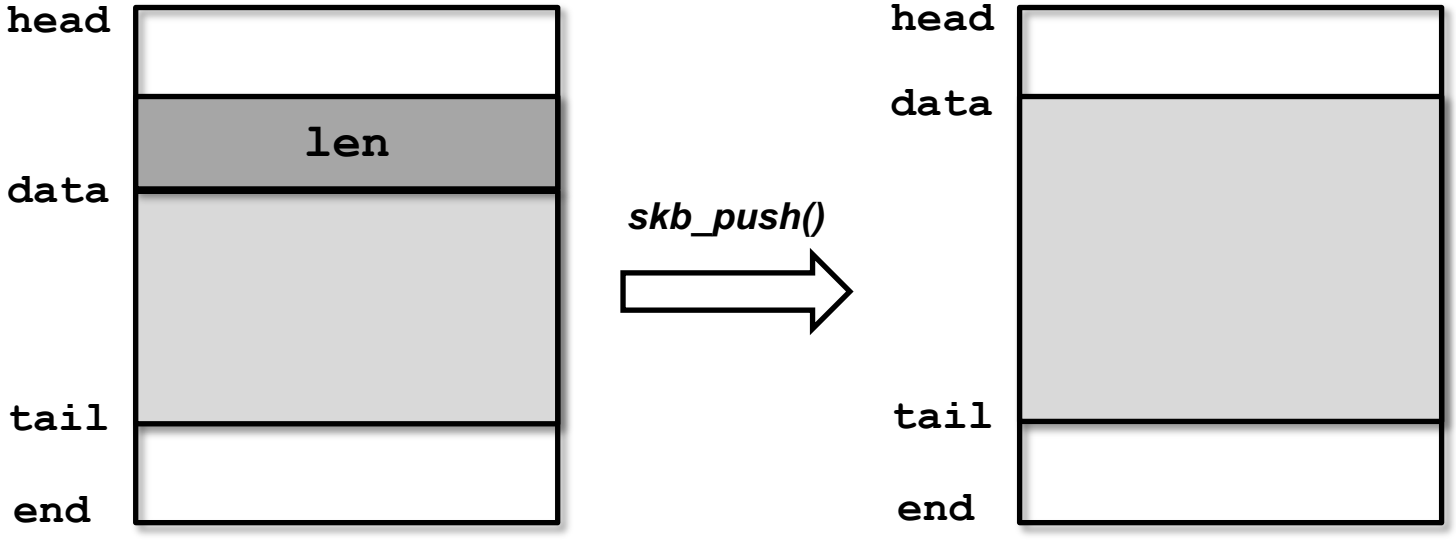
- **alloc_skb()**
  - GFP_ATOMIC：分配过程不能被中断，用于中断上下文中分配内存
  - GFP_KERNEL：分配过程可以被中断，分配请求被放到等待队列中
- **在上半部ISR中：**
  - GFP_ATOMIC保证分配过程不会再有ISR打断
- **在下半部软中断中：**
  - GFP_ATOMIC告诉内核，如果申请失败，不能进入睡眠

skb操作函数

- **unsigned char *skb_push(struct sk_buff *skb, unsigned int len)**
  - 在存储空间的头部增加存储网络报文的空间，**用于发送网络报文时添加包头**
- **unsigned char *skb_put(struct sk_buff *skb, unsigned int len)**
  - 在存储空间的尾部增加存储网络报文的空间，**用于发送网络报文时追加数据**
- **unsigned char *skb_pull(struct sk_buff *skb, unsigned int len)**
  - 使data指针指向下一层网络报文的头部，**用于接收网络报文时调整头部**
- **void skb_reserve(struct sk_buff *skb, unsigned int len)**
  - 在存储空间的头部预留len长度的空隙，**用于为协议头部保留空间**
- **void skb_trim(struct sk_buff *skb, unsigned int len)**
  - 将网络报文的长度缩减到*len*，用于丢弃网络报文尾部的填充值

skb_push

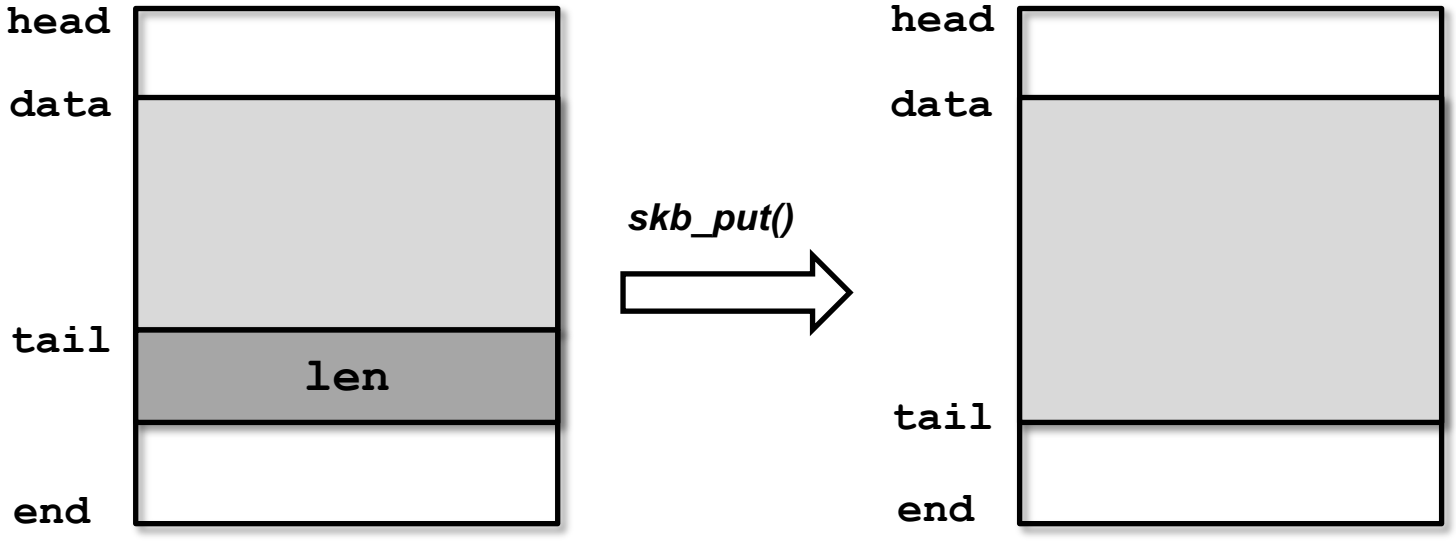
- 将skb的data指针往上推



- 用于TCP层或IP层封装头部

skb_put

- 将skb的tail指针往下移

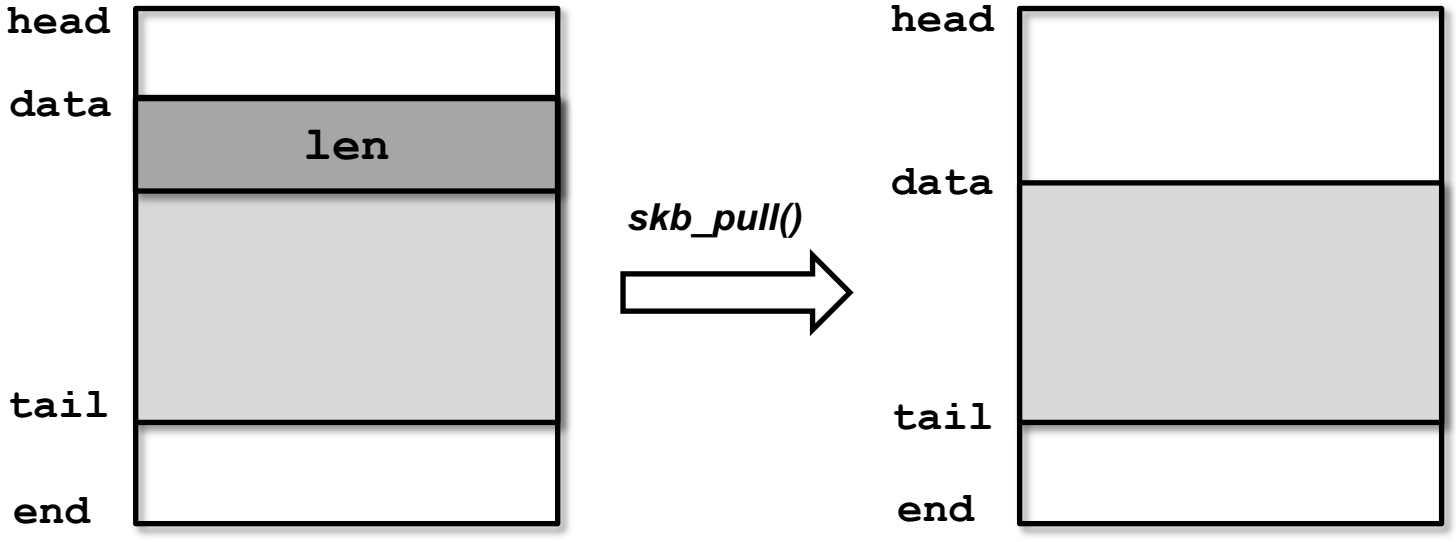


- 用于DMA接收数据包，或*copy_from_user*()发送数据包

skb_pull

注意：头部并没有从skb中删除！

- 将skb的data指针往下拉



- 用于IP层和TCP接收数据包时移除头部

skb总结

- **sk_buff**
  - 用于 Linux 网络子系统中各层之间的数据传递
  - 不同协议层的处理函数通过控制sk_buff结构来共享网络报文
- **收包：**
  - 网卡收到数据包后，将以太网帧数据转换为sk_buff数据结构
  - 各层剥去相应的协议头部，直至交给用户
- **发包：**
  - 网络模块必须建立一个包含待传输的数据包的sk_buff
  - 各层在sk_buff 中添加对应的协议头部直至交给网卡发送

用户态协议栈：DPDK

Linux网络协议栈的问题

- 网络设备的速度越来越高
- CPU朝着多核方向发展

- 内核协议栈逐渐成为高性能网络的**性能瓶颈**

Linux网络协议栈的问题

- **中断处理**
  - 大量网络包到来→频繁的硬件中断请求
  - 中断频繁打断较低优先级的软中断或者系统调用的执行过程→网络处理缓慢
- **上下文切换**
  - 线程间的调度产生频繁上下文切换开销
  - 锁竞争的开销严重：cacheline sharing
- **内存拷贝**
  - 数据从网卡DMA传到内核缓冲区，再从内核空间拷贝到用户空间
  - 占据Linux 内核协议栈数据包整个处理流程的 57.1%

Linux网络协议栈的问题

- **内存管理**
  - 内存页大小为4K，对TLB要求更高

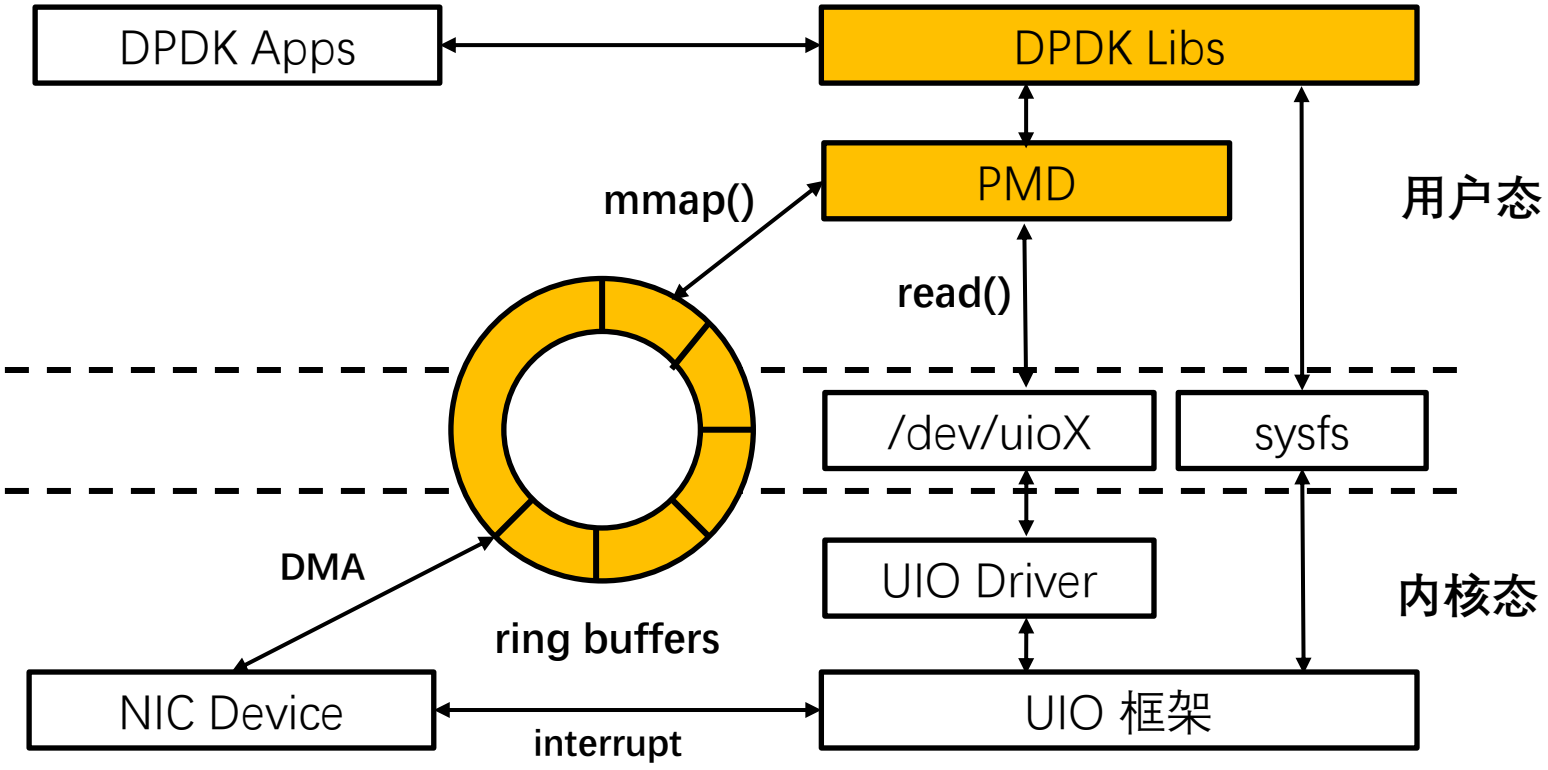
- **局部性失效**
  - 数据包处理可能跨多核：导致CPU 缓存失效，空间局部性差
  - NUMA 架构：存在跨 NUMA内存访问，性能影响很大

Intel DPDK

- **Data Plane Development Kit**
- **绕过Linux内核协议栈（bypass kernel）**
  - 直接在用户空间实现数据包的收发和处理

- **Linux User I/O**
  - 在用户态访问 MMIO（和设备交互）与DMA ring buffers
- **轮询模式驱动：**Poll Mode Driver (PMD)

DPDK架构图


DPDK Apps
DPDK Libs
PMD
mmap()
read()
用户态
/dev/uioX
sysfs
DMA
ring buffers
UIO Driver
内核态
NIC Device
interrupt
UIO 框架

DPDK特性

- **抛弃中断，使用轮询模式**
- **使用大页（2MB），减少TLB miss**

- **控制平面与数据平面相分离：**
  - 内核态负责“控制平面”：内核仅负责控制指令的处理
  - 用户态负责“数据平面”：将数据包处理、内存管理、处理器调度等任务转移到用户空间去完成，有效避免了繁重的模式切换

DPDK特性（2）

- **用多核编程代替多线程技术**
  - 绑核：设置 CPU 亲和性，减少彼此间调度切换
  - 无锁环形队列：解决资源竞争问题

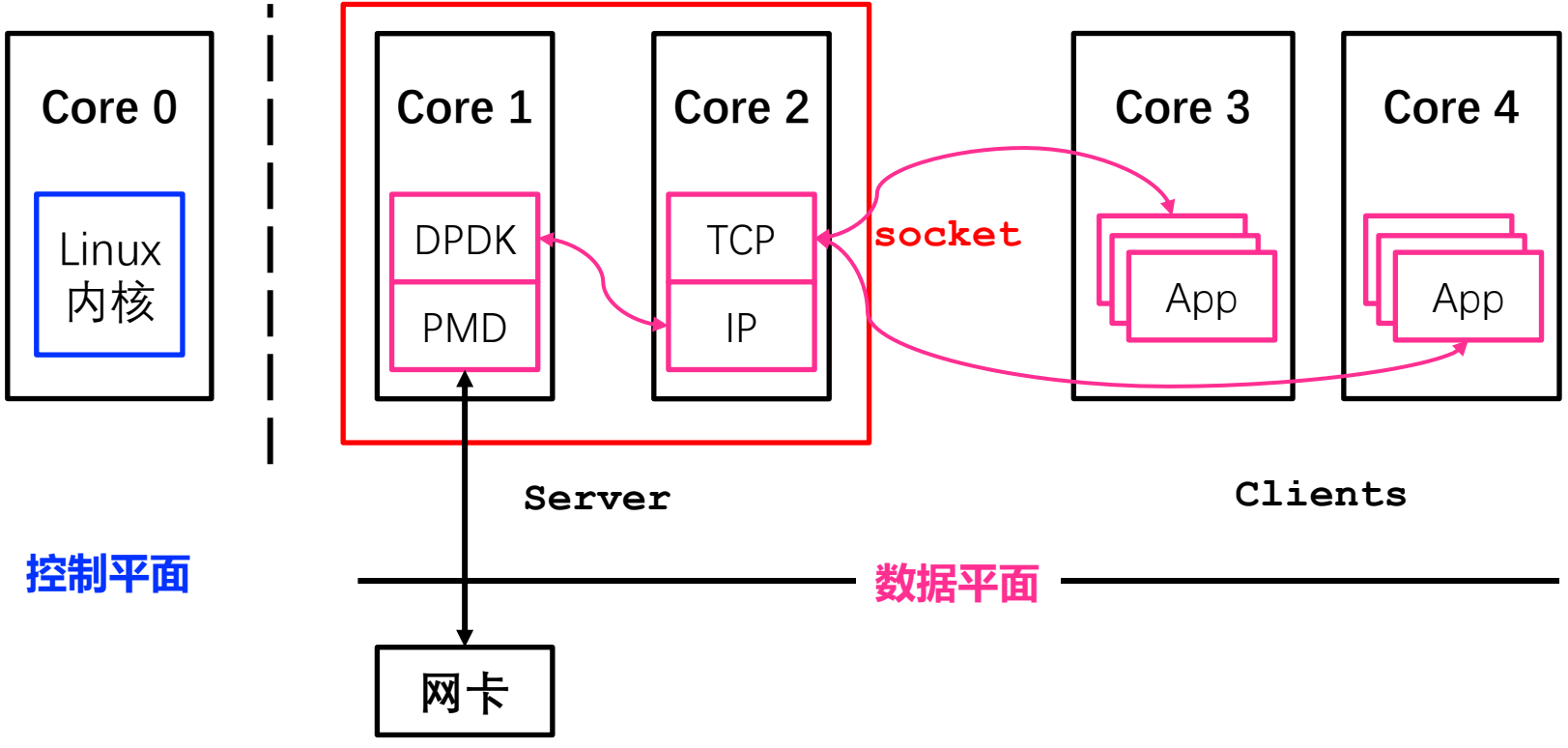
- **CPU 核尽量使用所在 NUMA 节点的内存**
  - 避免跨NUMA内存访问

DPDK的使用

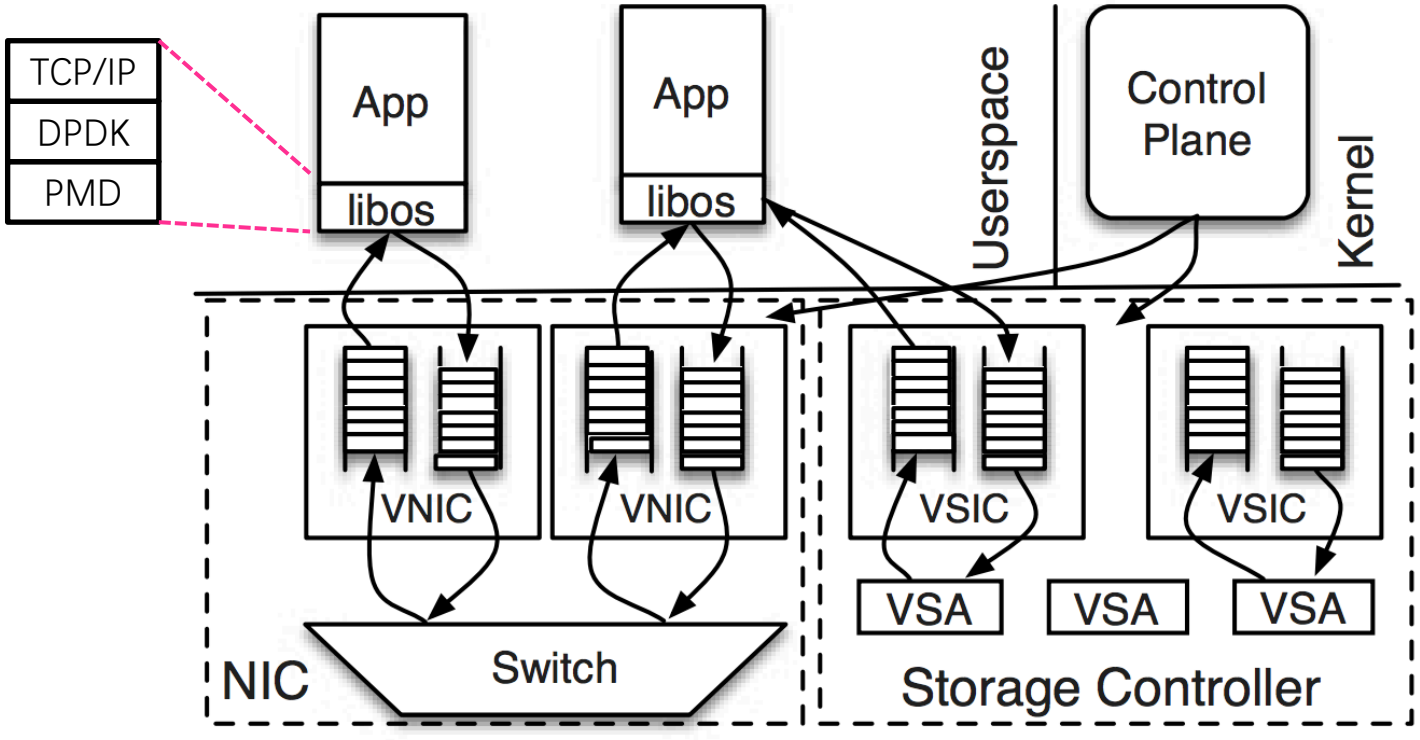
- **DPDK本身只是2层协议，不提供socket接口**
  - 适合于软路由场景

- **如果要用于应用程序，提供socket抽象**
  - **类微内核方案**：将 **DPDK+TCP/IP协议栈** 作为一个Server，为每个应用单独维护 socket fd 和进程间的对应关系
  - **LibOS方案**：需要逐个应用添加协议栈支持

类微内核方案

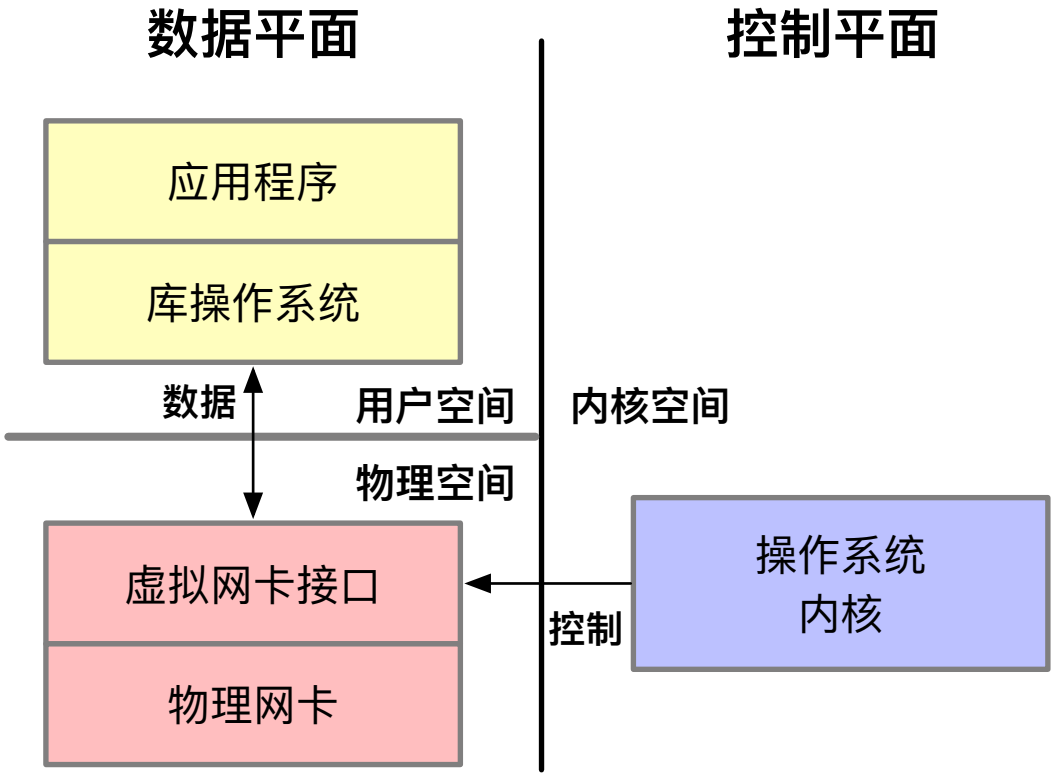

Core 0
Linux
内核
Core 1
DPDK
PMD
Core 2
TCP
IP
socket
Core 3
App
Core 4
App
Server
Clients
控制平面
数据平面
网卡

LibOS方案



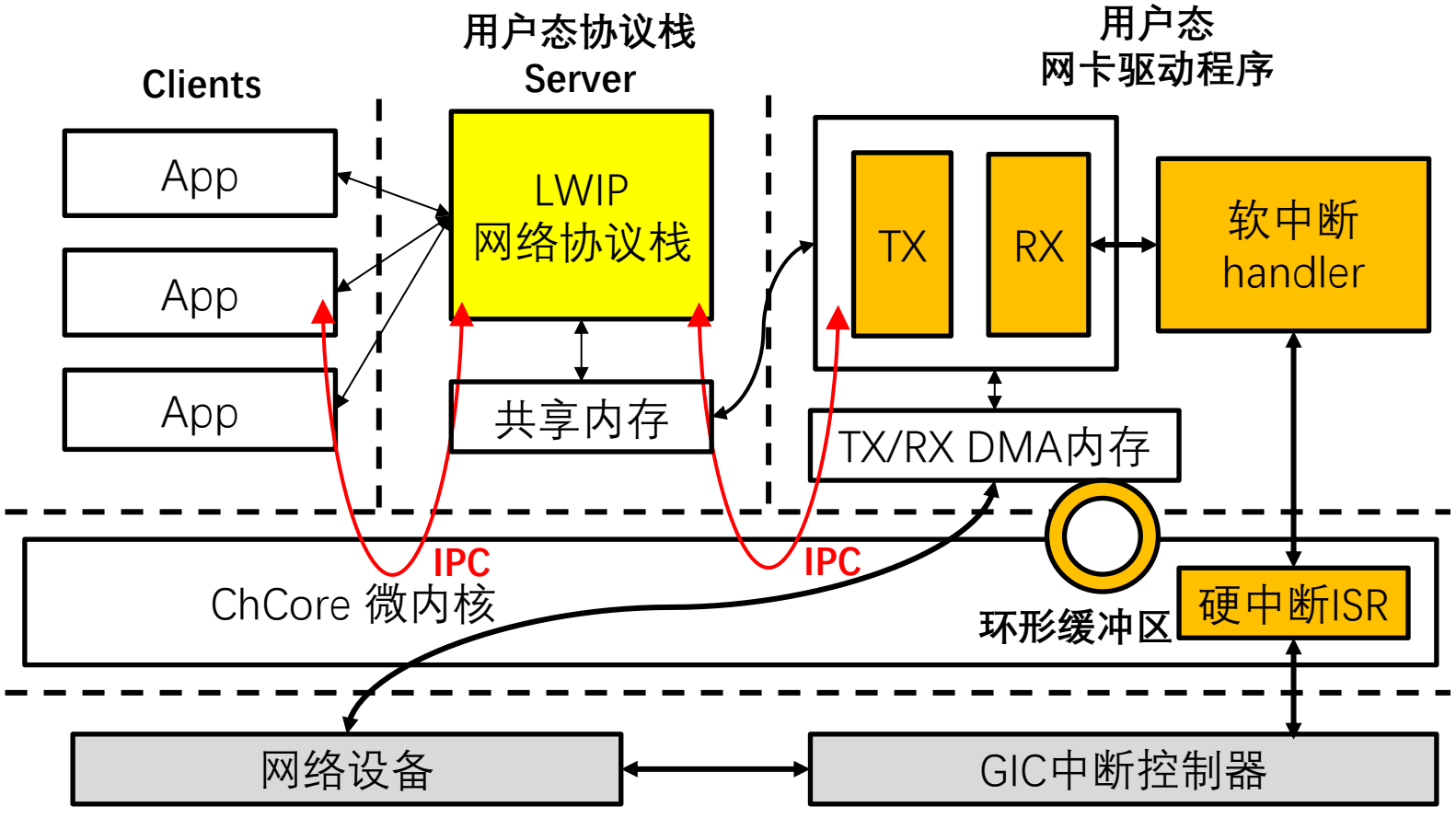
Arrakis: The Operating System is the Control Plane, OSDI 2014

控制平面与数据平面分离



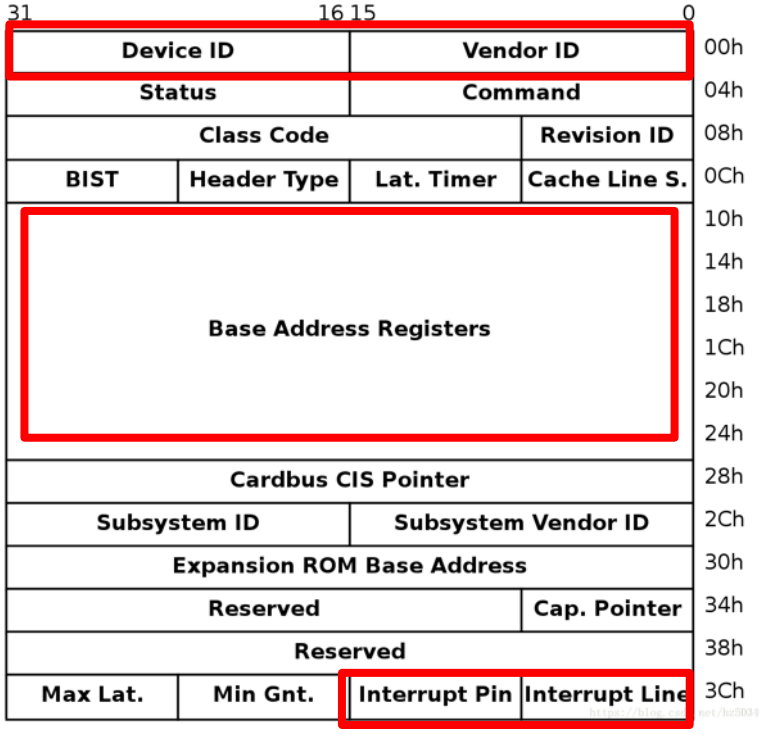
微内核协议栈： CHCORE

ChCore网络架构图



PCI设备

- **PCI总线为设备提供了标准机制**
  - 发现设备
  - 分配中断、物理内存空间和I/O空间
- **三层架构**
  - bus
  - device
  - function



找到目标网卡

```
struct pci_driver pci_attach_devs[] = { { 0x10ec, 0x8139, &rtl8139_attach }, };

static int pci_attach(struct pci_func *f) /* 根据vendor_id和product_id检测rtl8139网卡 */
{
    uint32_t vendor_id = PCI_VENDOR(f->dev_id);
    uint32_t product_id = PCI_PRODUCT(f->dev_id);
    for (int i = 0; i < sizeof(pci_attach_devs) / sizeof(pci_attach_devs[0]); i++)
        if (pci_attach_devs[i].vendor_id == vendor_id && pci_attach_devs[i].product_id == product_id)
            int r = pci_attach_devs[i].attachfn(f);
    return -ESUPPORT;
}

static void init_pci()
{
    static struct pci_bus root_bus = { 0 };
    f_iter_dev.bus = &root_bus;
    for (f_iter_dev.dev = 0; f_iter_dev.dev < 32; f_iter_dev.dev++) {
        ...
        intr = pci_conf_read(&f, PCI_INTERRUPT_REG);
        f.irq_line = PCI_INTERRUPT_LINE(intr); // 获取中断线
        f.dev_class = pci_conf_read(&f, PCI_CLASS_REG);
        pci_attach(&f);
    }
}
```

网卡初始化

```
struct rtl8139_dev_t
{
    uint64_t io_base, mem_base, irq_num;
    uint8_t  mac_addr[6];
    uint64_t rx_curr, tx_curr; // 缓冲区的当前偏移量
    void *   rx_buffer; // DMA缓冲区
    void *   tx_buffer;
} rtl8139_device;

static void rtl8139_initialize()
{
    // 初始化PCI配置空间
    rtl8139_device.io_base  = RTL8139_IO_VADDR & ~0xf;
    rtl8139_device.mem_base = RTL8139_MEM_VADDR;

    rtl8139_probe(); // 探测网卡设备

    // 申请 RX 缓冲区
    rtl8139_device.rx_buffer = rtl8139_kmalloc(RX_BUF_LEN, RTL8139_RX_DMA_VADDR, &kern_dma_addr);

    // 申请 TX 缓冲区
    rtl8139_device.tx_buffer = rtl8139_kmalloc(TX_BUF_SIZE, RTL8139_TX_DMA_VADDR, &kern_dma_addr);
}
```

用户态中断处理函数

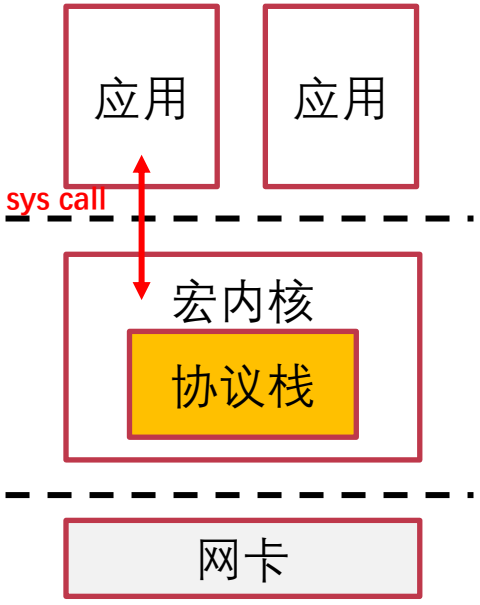
```
static void rtl8139_handler()
{
    uint16_t status = inports(rtl8139_device.io_base + 0x3e);
    outports(rtl8139_device.io_base + 0x3e, 0x5); // 向网卡确认中断

    if (status & ROK) {
        rtl8139_receive();  // 处理RX中断
    } else (status & TOK) ; // 判断TX中断
}

#define ARM_VIRT_NIC_IRQ    0x24
static void * nic_irq_handler(void * args)
{
    int irq_cap = usys_irq_register(ARM_VIRT_NIC_IRQ);
    while (1) {
        usys_irq_wait(irq_cap, true); // 等待ChCore内核向用户空间发送软中断
        rtl8139_handler();            // 网卡中断处理函数
        usys_irq_ack(irq_cap);        // 向ChCore确认，软中断已经收到
    }
    usys_exit(0);
    return NULL;
}
```

驱动和协议栈IPC交互

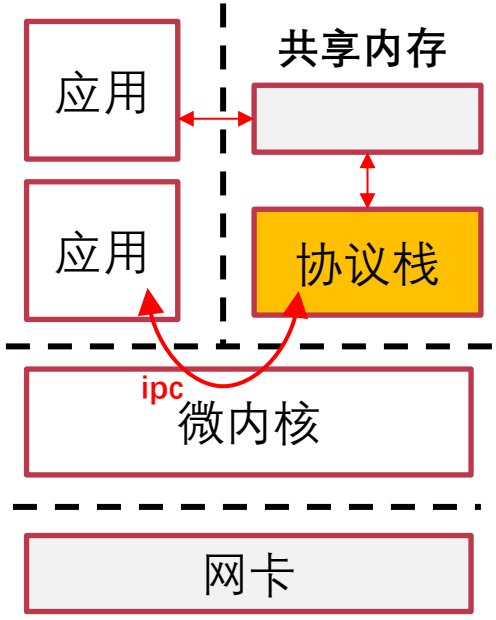
- **协议栈rtl8139_receive线程调用obtain_pkt()**
  - 将数据包组装成lwip的pbuf（类似Linux的skb）
  - 将pbuf加入共享内存的队列中
- **obtain_pkt ← IPC → lwip的RX线程**
  - lwip从队列中获取pbuf并进行解析，得到最终数据
- **lwip server线程 ← IPC → 用户态程序socket**
  - 用户程序使用socket，调用SYS_socket
  - SYS_socket被trap为IPC，从lwip server获取网络数据

不同架构对比

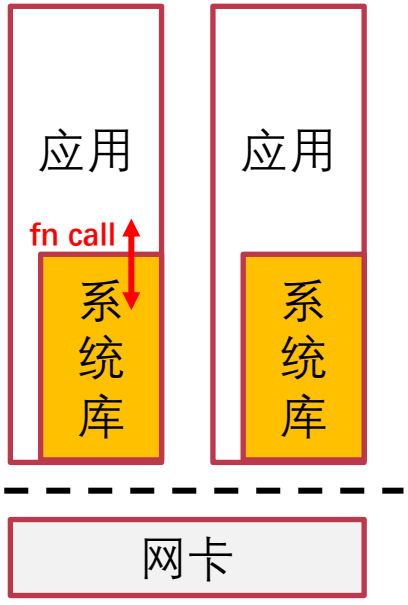
架构对比



宏内核系统



微内核系统



库操作系统

不同系统架构下网络设计对比

- **宏内核系统网络模块：**
  - 控制平面和数据平面都经过内核
  - 驱动和协议栈处在同一地址空间，没有模式切换
  - 驱动程序的安全问题会波及协议栈

不同系统架构下网络设计对比

- **微内核系统网络模块：**
  - 用户态驱动+协议栈，安全性好
  - 只有一个协议栈，运维成本低
  - 控制平面通信需要借助IPC完成，有一定开销

不同系统架构下网络设计对比

- **库操作系统（LibOS）网络模块：**
  - 用户态或non-root模式下协议栈，鲁棒性好
  - 每个实例都有自己的协议栈
  - 一旦需要更新，可维护性成本高
  - 多实例polling的情况下会导致浪费CPU*

Snap: a Microkernel Approach to Host Networking, SOSP 2019